```
::::::::::::::
Connections.hs
::::::::::::::
-- | Models the lattice of formulas.
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances,
             GeneralizedNewtypeDeriving, TupleSections #-}
module Connections where

import Control.Applicative
import Data.List
import Data.Map (Map,(!),keys,fromList,toList,mapKeys,elems,intersectionWith
                ,unionWith,singleton,foldrWithKey,assocs,mapWithKey
                ,filterWithKey,member)
import Data.Set (Set,isProperSubsetOf)
import qualified Data.Map as Map
import qualified Data.Set as Set
import Data.Maybe
import Test.QuickCheck

newtype Name = Name String
  deriving (Arbitrary,Eq,Ord)

instance Show Name where
  show (Name i) = i

swapName :: Name -> (Name,Name) -> Name
swapName k (i,j) | k == i    = j
                 | k == j    = i
                 | otherwise = k

-- | Directions
data Dir = Zero | One
  deriving (Eq,Ord)

instance Show Dir where
  show Zero = "0"
  show One  = "1"

instance Num Dir where
  Zero + Zero = Zero
  _    + _    = One

  Zero * _ = Zero
  One  * x = x

  abs      = id
  signum _ = One

  negate Zero = One
  negate One  = Zero

  fromInteger 0 = Zero
  fromInteger 1 = One
  fromInteger _ = error "fromInteger Dir"

instance Arbitrary Dir where
  arbitrary = do
    b <- arbitrary
    return $ if b then Zero else One

-- | Face

-- Faces of the form: [(i,0),(j,1),(k,0)]
type Face = Map Name Dir

instance {-# OVERLAPPING #-} Arbitrary Face where
  arbitrary = fromList <$> arbitrary

showFace :: Face -> String
```

```
showFace alpha = concat [ "(" ++ show i ++ " = " ++ show d ++ ")"
                        | (i,d) <- toList alpha ]

swapFace :: Face -> (Name,Name) -> Face
swapFace alpha ij = mapKeys (`swapName` ij) alpha

-- Check if two faces are compatible
compatible :: Face -> Face -> Bool
compatible xs ys = and (elems (intersectionWith (==) xs ys))

compatibles :: [Face] -> Bool
compatibles []     = True
compatibles (x:xs) = all (x `compatible`) xs && compatibles xs

allCompatible :: [Face] -> [(Face,Face)]
allCompatible []     = []
allCompatible (f:fs) = map (f,) (filter (compatible f) fs) ++ allCompatible fs

-- Partial composition operation
meet :: Face -> Face -> Face
meet = unionWith f
  where f d1 d2 = if d1 == d2 then d1 else error "meet: incompatible faces"

meetMaybe :: Face -> Face -> Maybe Face
meetMaybe x y = if compatible x y then Just $ meet x y else Nothing

-- | Quick check test that meet is commutative
meetCom :: Face -> Face -> Property
meetCom xs ys = compatible xs ys ==> xs `meet` ys == ys `meet` xs

-- | Quick check test that meet is associative
meetAssoc :: Face -> Face -> Face -> Property
meetAssoc xs ys zs = compatibles [xs,ys,zs] ==>
                     xs `meet` (ys `meet` zs) == (xs `meet` ys) `meet` zs

meetId :: Face -> Bool
meetId xs = xs `meet` xs == xs

-- | all meets of all compatible elements.
meets :: [Face] -> [Face] -> [Face]
meets xs ys = nub [ meet x y | x <- xs, y <- ys, compatible x y ]

meetss :: [[Face]] -> [Face]
meetss = foldr meets [eps]

-- | a <= b iff a /\ b = a. Recall that meet is partial.
leq :: Face -> Face -> Bool
alpha `leq` beta = meetMaybe alpha beta == Just alpha

comparable :: Face -> Face -> Bool
comparable alpha beta = alpha `leq` beta || beta `leq` alpha

-- | TODO: seems sensitive to ordering?
incomparables :: [Face] -> Bool
incomparables []     = True
incomparables (x:xs) = all (not . (x `comparable`)) xs && incomparables xs

(~>) :: Name -> Dir -> Face
i ~> d = singleton i d

eps :: Face
eps = Map.empty

minus :: Face -> Face -> Face
minus alpha beta = alpha Map.\\ beta

-- Compute the witness of A <= B, ie compute C s.t. B = CA
-- leqW :: Face -> Face -> Face
-- leqW = undefined
```

```haskell
-- | Logical Formulas over this algebra.
data Formula = Dir Dir
             | Atom Name
             | NegAtom Name
             | Formula :/\: Formula
             | Formula :\/: Formula
  deriving Eq

instance Show Formula where
  show (Dir Zero)  = "0"
  show (Dir One)   = "1"
  show (NegAtom a) = '-' : show a
  show (Atom a)    = show a
  show (a :\/: b)  = show1 a ++ " \\/ " ++ show1 b
    where show1 v@(a :/\: b) = "(" ++ show v ++ ")"
          show1 a = show a
  show (a :/\: b) = show1 a ++ " /\\ " ++ show1 b
    where show1 v@(a :\/: b) = "(" ++ show v ++ ")"
          show1 a = show a

arbFormula :: [Name] -> Int -> Gen Formula
arbFormula names s =
  frequency [ (1, Dir <$> arbitrary)
            , (1, Atom <$> elements names)
            , (1, NegAtom <$> elements names)
            , (s, do op <- elements [andFormula,orFormula]
                     op <$> arbFormula names s' <*> arbFormula names s')
            ]
  where s' = s `div` 3

instance Arbitrary Formula where
  arbitrary = do
      n <- arbitrary :: Gen Integer
      sized $ arbFormula (map (\x -> Name ('!' : show x))  [0..(abs n)])

class ToFormula a where
  toFormula :: a -> Formula

instance ToFormula Formula where
  toFormula = id

instance ToFormula Name where
  toFormula = Atom

instance ToFormula Dir where
  toFormula = Dir

negFormula :: Formula -> Formula
negFormula (Dir b)        = Dir (- b) -- exploit Num instance of Dir.
negFormula (Atom i)       = NegAtom i
negFormula (NegAtom i)    = Atom i
-- | use smart constructors orFormula, andFormula for constant folding.
negFormula (phi :/\: psi) = orFormula (negFormula phi) (negFormula psi)
negFormula (phi :\/: psi) = andFormula (negFormula phi) (negFormula psi)

andFormula :: Formula -> Formula -> Formula
andFormula (Dir Zero) _  = Dir Zero
andFormula _ (Dir Zero)  = Dir Zero
andFormula (Dir One) phi = phi
andFormula phi (Dir One) = phi
andFormula phi psi       = phi :/\: psi

orFormula :: Formula -> Formula -> Formula
orFormula (Dir One) _    = Dir One
orFormula _ (Dir One)    = Dir One
orFormula (Dir Zero) phi = phi
orFormula phi (Dir Zero) = phi
orFormula phi psi        = phi :\/: psi

-- | formula to sets of solutions (?)
```

```haskell
dnf :: Formula -> Set (Set (Name,Dir))
dnf (Dir One)      = Set.singleton Set.empty
dnf (Dir Zero)     = Set.empty
dnf (Atom n)       = Set.singleton (Set.singleton (n,1))
dnf (NegAtom n)    = Set.singleton (Set.singleton (n,0))
dnf (phi :\/: psi) = dnf phi `merge` dnf psi
dnf (phi :/\: psi) =
  foldr merge Set.empty [ Set.singleton (a `Set.union` b)
                        | a <- Set.toList (dnf phi)
                        , b <- Set.toList (dnf psi) ]

fromDNF :: Set (Set (Name,Dir)) -> Formula
fromDNF s = foldr (orFormula . foldr andFormula (Dir One)) (Dir Zero) fs
  where xss = map Set.toList $ Set.toList s
        fs = [ [ if d == Zero then NegAtom n else Atom n | (n,d) <- xs ] | xs <- xss ]

merge :: Set (Set (Name,Dir)) -> Set (Set (Name,Dir)) -> Set (Set (Name,Dir))
merge a b =
  let as = Set.toList a
      bs = Set.toList b
  in Set.fromList [ ai | ai <- as, not (any (`isProperSubsetOf` ai) bs) ] `Set.union`
     Set.fromList [ bi | bi <- bs, not (any (`isProperSubsetOf` bi) as) ]

-- evalFormula :: Formula -> Face -> Formula
-- evalFormula phi alpha =
--   Map.foldWithKey (\i d psi -> act psi (i,Dir d)) phi alpha

  -- (Dir b) alpha  = Dir b
-- evalFormula (Atom i) alpha = case Map.lookup i alpha of
--                                Just b -> Dir b
--                                Nothing -> Atom i
-- evalFormula (Not phi) alpha = negFormula (evalFormula phi alpha)
-- evalFormula (phi :/\: psi) alpha =
--   andFormula (evalFormula phi alpha) (evalFormula psi alpha)
-- evalFormula (phi :\/: psi) alpha =
--   orFormula (evalFormula phi alpha) (evalFormula psi alpha)

-- TODO: think about what the hell this means.
-- find a better name?
-- phi b = max {alpha : Face | phi alpha = b}
invFormula :: Formula -> Dir -> [Face]
invFormula (Dir b') b          = [ eps | b == b' ]
invFormula (Atom i) b          = [ singleton i b ]
invFormula (NegAtom i) b       = [ singleton i (- b) ]
invFormula (phi :/\: psi) Zero = invFormula phi 0 `union` invFormula psi 0
invFormula (phi :/\: psi) One  = meets (invFormula phi 1) (invFormula psi 1)
invFormula (phi :\/: psi) b    = invFormula (negFormula phi :/\: negFormula psi) (- b)

propInvFormulaIncomp :: Formula -> Dir -> Bool
propInvFormulaIncomp phi b = incomparables (invFormula phi b)

-- prop_invFormula :: Formula -> Dir -> Bool
-- prop_invFormula phi b =
--   all (\alpha -> phi `evalFormula` alpha == Dir b) (invFormula phi b)

-- testInvFormula :: [Face]
-- testInvFormula = invFormula (Atom (Name 0) :/\: Atom (Name 1)) 1

-- | Nominal

-- gensym :: [Name] -> Name
-- gensym xs = head (ys \\ xs)
--   where ys = map Name $ ["i","j","k","l"] ++ map (('i':) . show) [0..]

-- gensymNice :: Name -> [Name] -> Name
-- gensymNice i@(Name s) xs = head (ys \\ xs)
--   where ys = i:map (\n -> Name (s ++ show n)) [0..]

-- | Assumes names are named `!x`.
gensym :: [Name] -> Name
```

```haskell
gensym xs = Name ('!' : show max)
  where max = maximum' [ read x | Name ('!':x) <- xs ]
        maximum' [] = 0
        maximum' xs = maximum xs + 1

gensyms :: [Name] -> [Name]
gensyms d = let x = gensym d in x : gensyms (x : d)

class Nominal a where
  -- | return all names in play. aka support of the
  -- partial function [name -> value]
  support :: a -> [Name]
  -- | a[Name/Formula]. Substitute Name := Formula into a.
  act     :: a -> (Name,Formula) -> a
  -- | swap a (x, y) will swap occurences `x <-> y`.
  swap    :: a -> (Name,Name) -> a

fresh :: Nominal a => a -> Name
fresh = gensym . support

-- freshNice :: Nominal a => Name -> a -> Name
-- freshNice i = gensymNice i . support

freshs :: Nominal a => a -> [Name]
freshs = gensyms . support

unions :: Eq a => [[a]] -> [a]
unions = foldr union []

unionsMap :: Eq b => (a -> [b]) -> [a] -> [b]
unionsMap f = unions . map f

newtype Nameless a = Nameless { unNameless :: a }
                   deriving (Eq, Ord)

instance Nominal (Nameless a) where
  support _ = []
  act x _   = x
  swap x _  = x

instance Nominal () where
  support () = []
  act () _   = ()
  swap () _  = ()

instance (Nominal a, Nominal b) => Nominal (a, b) where
  support (a, b) = support a `union` support b
  act (a,b) f    = (act a f,act b f)
  swap (a,b) n   = (swap a n,swap b n)

instance (Nominal a, Nominal b, Nominal c) => Nominal (a, b, c) where
  support (a,b,c) = unions [support a, support b, support c]
  act (a,b,c) f   = (act a f,act b f,act c f)
  swap (a,b,c) n  = (swap a n,swap b n,swap c n)

instance (Nominal a, Nominal b, Nominal c, Nominal d) =>
         Nominal (a, b, c, d) where
  support (a,b,c,d) = unions [support a, support b, support c, support d]
  act (a,b,c,d) f   = (act a f,act b f,act c f,act d f)
  swap (a,b,c,d) n  = (swap a n,swap b n,swap c n,swap d n)

instance (Nominal a, Nominal b, Nominal c, Nominal d, Nominal e) =>
         Nominal (a, b, c, d, e) where
  support (a,b,c,d,e)  =
    unions [support a, support b, support c, support d, support e]
  act (a,b,c,d,e) f    = (act a f,act b f,act c f,act d f, act e f)
  swap (a,b,c,d,e) n =
    (swap a n,swap b n,swap c n,swap d n,swap e n)

instance (Nominal a, Nominal b, Nominal c, Nominal d, Nominal e, Nominal h) =>
```

```
         Nominal (a, b, c, d, e, h) where
  support (a,b,c,d,e,h) =
    unions [support a, support b, support c, support d, support e, support h]
  act (a,b,c,d,e,h) f   = (act a f,act b f,act c f,act d f, act e f, act h f)
  swap (a,b,c,d,e,h) n  =
    (swap a n,swap b n,swap c n,swap d n,swap e n,swap h n)

instance Nominal a => Nominal [a]  where
  support xs  = unions (map support xs)
  act xs f    = [ act x f | x <- xs ]
  swap xs n   = [ swap x n | x <- xs ]

instance Nominal a => Nominal (Maybe a)  where
  support    = maybe [] support
  act v f    = fmap (`act` f) v
  swap a n   = fmap (`swap` n) a

instance Nominal Formula where
  support (Dir _)        = []
  support (Atom i)       = [i]
  support (NegAtom i)    = [i]
  support (phi :/\: psi) = support phi `union` support psi
  support (phi :\/: psi) = support phi `union` support psi

  act (Dir b) (i,phi)  = Dir b
  act (Atom j) (i,phi) | i == j    = phi
                       | otherwise = Atom j
  act (NegAtom j) (i,phi) | i == j    = negFormula phi
                          | otherwise = NegAtom j
  act (psi1 :/\: psi2) (i,phi) = act psi1 (i,phi) `andFormula` act psi2 (i,phi)
  act (psi1 :\/: psi2) (i,phi) = act psi1 (i,phi) `orFormula` act psi2 (i,phi)

  swap (Dir b) (i,j)  = Dir b
  swap (Atom k) (i,j)| k == i    = Atom j
                     | k == j    = Atom i
                     | otherwise = Atom k
  swap (NegAtom k) (i,j)| k == i    = NegAtom j
                        | k == j    = NegAtom i
                        | otherwise = NegAtom k
  swap (psi1 :/\: psi2) (i,j) = swap psi1 (i,j) :/\: swap psi2 (i,j)
  swap (psi1 :\/: psi2) (i,j) = swap psi1 (i,j) :\/: swap psi2 (i,j)

face :: Nominal a => a -> Face -> a
face = foldrWithKey (\i d a -> act a (i,Dir d))

-- | the faces should be incomparable.
-- | "partial components" of a n-cube.
type System a = Map Face a

showListSystem :: Show a => [(Face,a)] -> String
showListSystem [] = "[]"
showListSystem ts =
  "[ " ++ intercalate ", " [ showFace alpha ++ " -> " ++ show u
                           | (alpha,u) <- ts ] ++ " ]"

showSystem :: Show a => System a -> String
showSystem = showListSystem . toList

-- | If face is <= any key, don't insert.
-- | Otherwise, insert face and remove all those faces which are <= this face.
-- | ie, can only "inflate" the system by adding larger keys, which then
-- | enforce their compatibility condition.
-- TODO: what about >= ?
insertSystem :: Face -> a -> System a -> System a
insertSystem alpha v ts
  | any (leq alpha) (keys ts) = ts
  | otherwise = Map.insert alpha v
                (Map.filterWithKey (\gamma _ -> not (gamma `leq` alpha)) ts)

insertsSystem :: [(Face, a)] -> System a -> System a
```

```haskell
insertsSystem faces us = foldr (uncurry insertSystem) us faces

mkSystem :: [(Face, a)] -> System a
mkSystem = flip insertsSystem Map.empty

unionSystem :: System a -> System a -> System a
unionSystem us vs = insertsSystem (assocs us) vs


-- | Monad?! WTF.
joinSystem :: System (System a) -> System a
joinSystem tss = mkSystem $
  [ (alpha `meet` beta,t) | (alpha,ts) <- assocs tss, (beta,t) <- assocs ts ]


-- TODO: add some checks
transposeSystemAndList :: System [a] -> [b] -> [(System a,b)]
transposeSystemAndList _  []      = []
transposeSystemAndList tss (u:us) =
  (Map.map head tss,u):transposeSystemAndList (Map.map tail tss) us

-- Quickcheck this:
-- (i = phi) * beta = (beta - i) * (i = phi beta)

-- Now we ensure that the keys are incomparable
instance Nominal a => Nominal (System a) where
  support s = unions (map keys $ keys s)
              `union` support (elems s)

  act s (i, phi) = addAssocs (assocs s)
    where
    addAssocs [] = Map.empty
    addAssocs ((alpha,u):alphaus) =
      let s' = addAssocs alphaus
      in case Map.lookup i alpha of
        Just d -> let beta = Map.delete i alpha
                  in foldr (\delta s'' -> insertSystem (meet delta beta)
                                            (face u (Map.delete i delta)) s'')
                                            s' (invFormula (face phi beta) d)
        Nothing -> insertSystem alpha (act u (i,face phi alpha)) s'

  swap s ij = mapKeys (`swapFace` ij) (Map.map (`swap` ij) s)

-- | TODO bollu: is border . border = 0? :)
-- | carve a using the same shape as the system b
border :: Nominal a => a -> System b -> System a
border v = mapWithKey (const . face v)

shape :: System a -> System ()
shape = border ()

instance {-# OVERLAPPING #-} (Nominal a, Arbitrary a) => Arbitrary (System a) where
  arbitrary = do
    a <- arbitrary
    border a <$> arbitraryShape (support a)
    where
      arbitraryShape :: [Name] -> Gen (System ())
      arbitraryShape supp = do
        phi <- sized $ arbFormula supp
        return $ fromList [(face,()) | face <- invFormula phi 0]

sym :: Nominal a => a -> Name -> a
sym a i = a `act` (i, NegAtom i)

rename :: Nominal a => a -> (Name, Name) -> a
rename a (i, j) = a `act` (i, Atom j)

conj, disj :: Nominal a => a -> (Name, Name) -> a
conj a (i, j) = a `act` (i, Atom i :/\: Atom j)
disj a (i, j) = a `act` (i, Atom i :\/: Atom j)
```

```
leqSystem :: Face -> System a -> Bool
alpha `leqSystem` us =
  not $ Map.null $ filterWithKey (\beta _ -> alpha `leq` beta) us

-- assumes alpha <= shape us
proj :: (Nominal a, Show a) => System a -> Face -> a
proj us alpha | eps `member` usalpha = usalpha ! eps
              | otherwise           =
  error $ "proj: eps not in " ++ show usalpha ++ "\nwhich  is the "
    ++ show alpha ++ "\nface of " ++ show us
  where usalpha = us `face` alpha

domain :: System a -> [Name]
domain  = keys . Map.unions . keys
::::::::::::::
CTT.hs
::::::::::::::
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}
module CTT where

import Control.Applicative
import Data.List
import Data.Maybe
import Data.Map (Map,(!),filterWithKey,elems)
import qualified Data.Map as Map
import Text.PrettyPrint as PP
import Data.Set (Set)
import qualified Data.Set as Set
import Prelude hiding ((<>))

import qualified Connections as C

--------------------------------------------------------------------------------
-- | Terms, with type |Ter|.
-- | General conventions: O for object, P for path.


-- | File locations
data Loc = Loc { locFile :: String
               , locPos  :: (Int,Int) }
  deriving Eq

type Ident  = String
-- | TODO: Identifier of Telescopes. Are of two types, Object and path based.
type LIdent = String

-- Telescope (x1 : A1) .. (xn : An)
type Tele   = [(Ident,Ter)]

-- | from Exp.cf:
-- |   C.System.     C.System ::= "[" [Side] "]" ;|
-- |   Side.   Side ::= [Face] "->" Exp ;
-- |   separator Side "," ;
-- |   Face.     Face ::= "(" AIdent "=" Dir ")" ;
-- |   separator Face "" ;
-- | C.System comes from Connections.hs.
-- | C.System Ter is a map from Face to Term,
--       where the faces are incomparable.
data Label = OLabel LIdent Tele -- Object label
           | PLabel LIdent Tele [C.Name] (C.System Ter) -- Path label
  deriving (Eq,Show)

-- | OBranch of the form: c x1 .. xn -> e
-- | PBranch of the form: c x1 .. xn i1 .. im -> e
data Branch = OBranch LIdent [Ident] Ter
            | PBranch LIdent [Ident] [C.Name] Ter
  deriving (Eq,Show)

-- Declarations: x : A = e
```

```haskell
-- A group of mutual declarations is identified by its location. It is used to
-- speed up the Eq instance for Ctxt.
type Decl  = (Ident,(Ter,Ter))
data Decls = MutualDecls Loc [Decl]
           | OpaqueDecl Ident
           | TransparentDecl Ident
           | TransparentAllDecl
           deriving Eq

declIdents :: [Decl] -> [Ident]
declIdents decls = [ x | (x,_) <- decls ]

declTers :: [Decl] -> [Ter]
declTers decls = [ d | (_,(_,d)) <- decls ]

-- | convert a sequence of declarations into a sequence of (ident: type)
declTele :: [Decl] -> Tele
declTele decls = [ (x,t) | (x,(t,_)) <- decls ]

declDefs :: [Decl] -> [(Ident,Ter)]
declDefs decls = [ (x,d) | (x,(_,d)) <- decls ]

labelTele :: Label -> (LIdent,Tele)
labelTele (OLabel c ts) = (c,ts)
labelTele (PLabel c ts _ _) = (c,ts)

labelName :: Label -> LIdent
labelName = fst . labelTele

labelTeles :: [Label] -> [(LIdent,Tele)]
labelTeles = map labelTele

lookupLabel :: LIdent -> [Label] -> Maybe Tele
lookupLabel x xs = lookup x (labelTeles xs)

lookupPLabel :: LIdent -> [Label] -> Maybe (Tele,[C.Name],C.System Ter)
lookupPLabel x xs = listToMaybe [ (ts,is,es) | PLabel y ts is es <- xs, x == y ]

branchName :: Branch -> LIdent
branchName (OBranch c _ _) = c
branchName (PBranch c _ _ _) = c

lookupBranch :: LIdent -> [Branch] -> Maybe Branch
lookupBranch _ []      = Nothing
lookupBranch x (b:brs) = case b of
  OBranch c _ _   | x == c    -> Just b
                  | otherwise -> lookupBranch x brs
  PBranch c _ _ _ | x == c    -> Just b
                  | otherwise -> lookupBranch x brs

-- TODO: Term v/s Value?
-- Terms
data Ter = Pi Ter -- TODO: ?
         | App Ter Ter -- f x
         | Lam Ident Ter Ter -- \x: T. e
         | Where Ter Decls -- TODO: ?
         | Var Ident -- x
         | U -- Unit
           -- Sigma types:
         | Sigma Ter -- TODO: ?
         | Pair Ter Ter -- (a, b)
         | Fst Ter -- fst t
         | Snd Ter -- snd t
           -- constructor c Ms
         | Con LIdent [Ter]
         | PCon LIdent Ter [Ter] [C.Formula] -- c A ts phis (A is the data type)
           -- branches c1 xs1  -> M1,..., cn xsn -> Mn
         | Split Ident Loc Ter [Branch]
           -- labelled sum c1 A1s,..., cn Ans (assumes terms are constructors)
         | Sum Loc Ident [Label] -- TODO: should only contain OLabels
```

```haskell
         | HSum Loc Ident [Label]
           -- undefined and holes
         | Undef Loc Ter -- Location and type
         | Hole Loc
           -- Path types
         | PathP Ter Ter Ter
         | PLam C.Name Ter
         | AppFormula Ter C.Formula
           -- Kan composition and filling
         | Comp Ter Ter (C.System Ter)
         | Fill Ter Ter (C.System Ter)
         | HComp Ter Ter (C.System Ter)
         -- Glue
         | Glue Ter (C.System Ter)
         | GlueElem Ter (C.System Ter)
         | UnGlueElem Ter (C.System Ter)
           -- Id
         | Id Ter Ter Ter
         | IdPair Ter (C.System Ter)
         | IdJ Ter Ter Ter Ter Ter Ter
  deriving Eq

-- For an expression t, returns (u,ts) where u is no application and t = u ts
unApps :: Ter -> (Ter,[Ter])
unApps = aux []
  where aux :: [Ter] -> Ter -> (Ter,[Ter])
        aux acc (App r s) = aux (s:acc) r
        aux acc t         = (t,acc)

mkApps :: Ter -> [Ter] -> Ter
mkApps (Con l us) vs = Con l (us ++ vs)
mkApps t ts          = foldl App t ts

mkWheres :: [Decls] -> Ter -> Ter
mkWheres []     e = e
mkWheres (d:ds) e = Where (mkWheres ds e) d

--------------------------------------------------------------------------------
-- | Values

data Val = VU
         | Ter Ter Env
         | VPi Val Val
         | VSigma Val Val
         | VPair Val Val
         | VCon LIdent [Val]
         | VPCon LIdent Val [Val] [C.Formula]

           -- Path values
         | VPathP Val Val Val
         | VPLam C.Name Val
         | VComp Val Val (C.System Val)

           -- Glue values
         | VGlue Val (C.System Val)
         | VGlueElem Val (C.System Val)
         | VUnGlueElem Val (C.System Val)

           -- Composition in the universe
         | VCompU Val (C.System Val)

           -- Composition for HITs; the type is constant
         | VHComp Val Val (C.System Val)

           -- Id
         | VId Val Val Val
         | VIdPair Val (C.System Val)

           -- TODO: Neutral => normalization by evaluation?
           -- Neutral values:
```

```haskell
         | VVar Ident Val
         | VOpaque Ident Val
         | VFst Val
         | VSnd Val
         | VSplit Val Val
         | VApp Val Val
         | VAppFormula Val C.Formula
         | VLam Ident Val Val
         | VUnGlueElemU Val Val (C.System Val)
         | VIdJ Val Val Val Val Val Val
  deriving Eq

isNeutral :: Val -> Bool
isNeutral v = case v of
  Ter Undef{} _  -> True
  Ter Hole{} _   -> True
  VVar{}         -> True
  VOpaque{}      -> True
  VComp{}        -> True
  VFst{}         -> True
  VSnd{}         -> True
  VSplit{}       -> True
  VApp{}         -> True
  VAppFormula{}  -> True
  VUnGlueElemU{} -> True
  VUnGlueElem{}  -> True
  VIdJ{}         -> True
  _              -> False

isNeutralSystem :: C.System Val -> Bool
isNeutralSystem = any isNeutral . elems

-- isNeutralPath :: Val -> Bool
-- isNeutralPath (VPath _ v) = isNeutral v
-- isNeutralPath _ = True

mkVar :: Int -> String -> Val -> Val
mkVar k x = VVar (x ++ show k)

mkVarNice :: [String] -> String -> Val -> Val
mkVarNice xs x = VVar (head (ys \\ xs))
  where ys = x:map (\n -> x ++ show n) [0..]

unCon :: Val -> [Val]
unCon (VCon _ vs) = vs
unCon v           = error $ "unCon: not a constructor: " ++ show v

isCon :: Val -> Bool
isCon VCon{} = True
isCon _      = False

-- Constant path: <_> v
constPath :: Val -> Val
constPath = VPLam (C.Name "_")


--------------------------------------------------------------------------------
-- | Environments

data Ctxt = Empty
          | Upd Ident Ctxt
          | Sub C.Name Ctxt
          | Def Loc [Decl] Ctxt
  deriving (Show)

instance Eq Ctxt where
    c == d = case (c, d) of
        (Empty, Empty)              -> True
        (Upd x c', Upd y d')        -> x == y && c' == d'
        (Sub i c', Sub j d')        -> i == j && c' == d'
```

```
        (Def m xs c', Def n ys d')  -> (m == n || xs == ys) && c' == d'
            -- Invariant: if two declaration groups come from the same
            -- location, they are equal and their contents are not compared.
        _                           -> False

-- The Idents and Names in the Ctxt refer to the elements in the two
-- lists. This is more efficient because acting on an environment now
-- only need to affect the lists and not the whole context.
-- The last list is the list of opaque names
-- | C.Nameless comes from Connections.hs
newtype Env = Env (Ctxt,[Val],[C.Formula],C.Nameless (Set Ident))
  deriving (Eq)

emptyEnv :: Env
emptyEnv = Env (Empty,[],[],C.Nameless Set.empty)

def :: Decls -> Env -> Env
def (MutualDecls m ds) (Env (rho,vs,fs,C.Nameless os)) = Env (Def m ds rho,vs,fs,C.Nameless (os Set.\\ Set.fromList (declIdents ds)))
def (OpaqueDecl n) (Env (rho,vs,fs,C.Nameless os)) = Env (rho,vs,fs,C.Nameless (Set.insert n os))
def (TransparentDecl n) (Env (rho,vs,fs,C.Nameless os)) = Env (rho,vs,fs,C.Nameless (Set.delete n os))
def TransparentAllDecl (Env (rho,vs,fs,C.Nameless os)) = Env (rho,vs,fs,C.Nameless Set.empty)

defWhere :: Decls -> Env -> Env
defWhere (MutualDecls m ds) (Env (rho,vs,fs,C.Nameless os)) = Env (Def m ds rho,vs,fs,C.Nameless (os Set.\\ Set.fromList (declIdents ds)))
defWhere (OpaqueDecl _) rho = rho
defWhere (TransparentDecl _) rho = rho
defWhere TransparentAllDecl rho = rho

sub :: (C.Name,C.Formula) -> Env -> Env
sub (i,phi) (Env (rho,vs,fs,os)) = Env (Sub i rho,vs,phi:fs,os)

upd :: (Ident,Val) -> Env -> Env
upd (x,v) (Env (rho,vs,fs,C.Nameless os)) = Env (Upd x rho,v:vs,fs,C.Nameless (Set.delete x os))

upds :: [(Ident,Val)] -> Env -> Env
upds xus rho = foldl (flip upd) rho xus

updsTele :: Tele -> [Val] -> Env -> Env
updsTele tele vs = upds (zip (map fst tele) vs)

subs :: [(C.Name,C.Formula)] -> Env -> Env
subs iphis rho = foldl (flip sub) rho iphis

mapEnv :: (Val -> Val) -> (C.Formula -> C.Formula) -> Env -> Env
mapEnv f g (Env (rho,vs,fs,os)) = Env (rho,map f vs,map g fs,os)

valAndFormulaOfEnv :: Env -> ([Val],[C.Formula])
valAndFormulaOfEnv (Env (_,vs,fs,_)) = (vs,fs)

valOfEnv :: Env -> [Val]
valOfEnv = fst . valAndFormulaOfEnv

formulaOfEnv :: Env -> [C.Formula]
formulaOfEnv = snd . valAndFormulaOfEnv

domainEnv :: Env -> [C.Name]
domainEnv (Env (rho,_,_,_)) = domCtxt rho
  where domCtxt rho = case rho of
          Empty      -> []
          Upd _ e    -> domCtxt e
          Def _ ts e -> domCtxt e
          Sub i e    -> i : domCtxt e

-- | Extract the context from the environment, used when printing holes
```

```haskell
contextOfEnv :: Env -> [String]
contextOfEnv rho = case rho of
  Env (Empty,_,_,_)              -> []
  Env (Upd x e,VVar n t:vs,fs,os) -> (n ++ " : " ++ show t) : contextOfEnv (Env (e,vs,fs,os))
  Env (Upd x e,v:vs,fs,os)        -> (x ++ " = " ++ show v) : contextOfEnv (Env (e,vs,fs,os))
  Env (Def _ _ e,vs,fs,os)        -> contextOfEnv (Env (e,vs,fs,os))
  Env (Sub i e,vs,phi:fs,os)      -> (show i ++ " = " ++ show phi) : contextOfEnv (Env (e,vs,fs,os))

--------------------------------------------------------------------------------
-- | Pretty printing

instance Show Env where
  show = render . showEnv True

showEnv :: Bool -> Env -> Doc
showEnv b e =
  let -- This decides if we should print "x = " or not
      names x = if b then text x <+> equals else PP.empty
      par   x = if b then parens x else x
      com     = if b then comma else PP.empty
      showEnv1 e = case e of
        Env (Upd x env,u:us,fs,os)   ->
          showEnv1 (Env (env,us,fs,os)) <+> names x <+> showVal1 u <> com
        Env (Sub i env,us,phi:fs,os) ->
          showEnv1 (Env (env,us,fs,os)) <+> names (show i) <+> text (show phi) <> com
        Env (Def _ _ env,vs,fs,os)   -> showEnv1 (Env (env,vs,fs,os))
        _                            -> showEnv b e
  in case e of
    Env (Empty,_,_,_)            -> PP.empty
    Env (Def _ _ env,vs,fs,os)   -> showEnv b (Env (env,vs,fs,os))
    Env (Upd x env,u:us,fs,os)   ->
      par $ showEnv1 (Env (env,us,fs,os)) <+> names x <+> showVal1 u
    Env (Sub i env,us,phi:fs,os) ->
      par $ showEnv1 (Env (env,us,fs,os)) <+> names (show i) <+> text (show phi)

instance Show Loc where
  show = render . showLoc

showLoc :: Loc -> Doc
showLoc (Loc name (i,j)) = text (show (i,j) ++ " in " ++ name)

showFormula :: C.Formula -> Doc
showFormula phi = case phi of
  _ C.:\/: _ -> parens (text (show phi))
  _ C.:/\: _ -> parens (text (show phi))
  _ -> text $ show phi

instance Show Ter where
  show = render . showTer

showTer :: Ter -> Doc
showTer v = case v of
  U                  -> char 'U'
  App e0 e1          -> showTer e0 <+> showTer1 e1
  Pi e0              -> text "Pi" <+> showTer e0
  Lam x t e          -> char '\\' <> parens (text x <+> colon <+> showTer t) <+>
                          text "->" <+> showTer e
  Fst e              -> showTer1 e <> text ".1"
  Snd e              -> showTer1 e <> text ".2"
  Sigma e0           -> text "Sigma" <+> showTer1 e0
  Pair e0 e1         -> parens (showTer e0 <> comma <> showTer e1)
  Where e d          -> showTer e <+> text "where" <+> showDecls d
  Var x              -> text x
  Con c es           -> text c <+> showTers es
  PCon c a es phis   -> text c <+> braces (showTer a) <+> showTers es
                        <+> hsep (map ((char '@' <+>) . showFormula) phis)
  Split f _ _ _      -> text f
```

```haskell
  Sum _ n _          -> text n
  HSum _ n _         -> text n
  Undef{}            -> text "undefined"
  Hole{}             -> text "?"
  PathP e0 e1 e2     -> text "PathP" <+> showTers [e0,e1,e2]
  PLam i e           -> char '<' <> text (show i) <> char '>' <+> showTer e
  AppFormula e phi   -> showTer1 e <+> char '@' <+> showFormula phi
  Comp e t ts        -> text "comp" <+> showTers [e,t] <+> text (C.showSystem ts)
  HComp e t ts       -> text "hComp" <+> showTers [e,t] <+> text (C.showSystem ts)
  Fill e t ts        -> text "fill" <+> showTers [e,t] <+> text (C.showSystem ts)
  Glue a ts          -> text "Glue" <+> showTer1 a <+> text (C.showSystem ts)
  GlueElem a ts      -> text "glue" <+> showTer1 a <+> text (C.showSystem ts)
  UnGlueElem a ts    -> text "unglue" <+> showTer1 a <+> text (C.showSystem ts)
  Id a u v           -> text "Id" <+> showTers [a,u,v]
  IdPair b ts        -> text "idC" <+> showTer1 b <+> text (C.showSystem ts)
  IdJ a t c d x p    -> text "idJ" <+> showTers [a,t,c,d,x,p]

showTers :: [Ter] -> Doc
showTers = hsep . map showTer1

showTer1 :: Ter -> Doc
showTer1 t = case t of
  U        -> char 'U'
  Con c [] -> text c
  Var{}    -> showTer t
  Undef{}  -> showTer t
  Hole{}   -> showTer t
  Split{}  -> showTer t
  Sum{}    -> showTer t
  HSum{}   -> showTer t
  Fst{}    -> showTer t
  Snd{}    -> showTer t
  _        -> parens (showTer t)

showDecls :: Decls -> Doc
showDecls (MutualDecls _ defs) =
  hsep $ punctuate comma
  [ text x <+> equals <+> showTer d | (x,(_,d)) <- defs ]
showDecls (OpaqueDecl i) = text "opaque" <+> text i
showDecls (TransparentDecl i) = text "transparent" <+> text i
showDecls TransparentAllDecl = text "transparent_all"

instance Show Val where
  show = render . showVal

showVal :: Val -> Doc
showVal v = case v of
  VU                -> char 'U'
  Ter t@Sum{} rho   -> showTer t <+> showEnv False rho
  Ter t@HSum{} rho  -> showTer t <+> showEnv False rho
  Ter t@Split{} rho -> showTer t <+> showEnv False rho
  Ter t rho         -> showTer1 t <+> showEnv True rho
  VCon c us         -> text c <+> showVals us
  VPCon c a us phis -> text c <+> braces (showVal a) <+> showVals us
                       <+> hsep (map ((char '@' <+>) . showFormula) phis)
  VHComp v0 v1 vs   -> text "hComp" <+> showVals [v0,v1] <+> text (C.showSystem vs)
  VPi a l@(VLam x t b)
    | "_" `isPrefixOf` x -> showVal1 a <+> text "->" <+> showVal1 b
    | otherwise          -> char '(' <> showLam v
  VPi a b           -> text "Pi" <+> showVals [a,b]
  VPair u v         -> parens (showVal u <> comma <> showVal v)
  VSigma u v        -> text "Sigma" <+> showVals [u,v]
  VApp u v          -> showVal u <+> showVal1 v
  VLam{}            -> text "\\(" <> showLam v
  VPLam{}           -> char '<' <> showPLam v
  VSplit u v        -> showVal u <+> showVal1 v
  VVar x _          -> text x
  VOpaque x _       -> text ('#':x)
  VFst u            -> showVal1 u <> text ".1"
  VSnd u            -> showVal1 u <> text ".2"
```

```haskell
  VPathP v0 v1 v2   -> text "PathP" <+> showVals [v0,v1,v2]
  VAppFormula v phi -> showVal v <+> char '@' <+> showFormula phi
  VComp v0 v1 vs    ->
    text "comp" <+> showVals [v0,v1] <+> text (C.showSystem vs)
  VGlue a ts        -> text "Glue" <+> showVal1 a <+> text (C.showSystem ts)
  VGlueElem a ts    -> text "glue" <+> showVal1 a <+> text (C.showSystem ts)
  VUnGlueElem a ts  -> text "unglue" <+> showVal1 a <+> text (C.showSystem ts)
  VUnGlueElemU v b es -> text "unglue U" <+> showVals [v,b]
                         <+> text (C.showSystem es)
  VCompU a ts       -> text "comp (<_> U)" <+> showVal1 a <+> text (C.showSystem ts)
  VId a u v           -> text "Id" <+> showVals [a,u,v]
  VIdPair b ts        -> text "idC" <+> showVal1 b <+> text (C.showSystem ts)
  VIdJ a t c d x p    -> text "idJ" <+> showVals [a,t,c,d,x,p]

showPLam :: Val -> Doc
showPLam e = case e of
  VPLam i a@VPLam{} -> text (show i) <+> showPLam a
  VPLam i a         -> text (show i) <> char '>' <+> showVal a
  _                 -> showVal e

-- Merge lambdas of the same type
showLam :: Val -> Doc
showLam e = case e of
  VLam x t a@(VLam _ t' _)
    | t == t'   -> text x <+> showLam a
    | otherwise ->
      text x <+> colon <+> showVal t <> char ')' <+> text "->" <+> showVal a
  VPi _ (VLam x t a@(VPi _ (VLam _ t' _)))
    | t == t'   -> text x <+> showLam a
    | otherwise ->
      text x <+> colon <+> showVal t <> char ')' <+> text "->" <+> showVal a
  VLam x t e         ->
    text x <+> colon <+> showVal t <> char ')' <+> text "->" <+> showVal e
  VPi _ (VLam x t e) ->
    text x <+> colon <+> showVal t <> char ')' <+> text "->" <+> showVal e
  _ -> showVal e

showVal1 :: Val -> Doc
showVal1 v = case v of
  VU                -> showVal v
  VCon c []         -> showVal v
  VVar{}            -> showVal v
  VFst{}            -> showVal v
  VSnd{}            -> showVal v
  Ter t rho | isEmpty (showEnv False rho) -> showTer1 t
  _                 -> parens (showVal v)

showVals :: [Val] -> Doc
showVals = hsep . map showVal1
::::::::::::::
Eval.hs
::::::::::::::
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}
module Eval where

import Data.List
import Data.Maybe (fromMaybe)
import Data.Map (Map,(!),mapWithKey,assocs,filterWithKey
                ,elems,intersectionWith,intersection,keys
                ,member,notMember,empty)
import qualified Data.Map as Map
import qualified Data.Set as Set

import qualified Connections as C
import CTT

-----------------------------------------------------------------------
-- Lookup functions

look :: String -> Env -> Val
```

```haskell
look x (Env (Upd y rho,v:vs,fs,os)) | x == y = v
                                    | otherwise = look x (Env (rho,vs,fs,os))
look x r@(Env (Def _ decls rho,vs,fs,C.Nameless os)) = case lookup x decls of
  Just (_,t) -> eval r t
  Nothing    -> look x (Env (rho,vs,fs,C.Nameless os))
look x (Env (Sub _ rho,vs,_:fs,os)) = look x (Env (rho,vs,fs,os))
look x (Env (Empty,_,_,_)) = error $ "look: not found " ++ show x

lookType :: String -> Env -> Val
lookType x (Env (Upd y rho,v:vs,fs,os))
  | x /= y        = lookType x (Env (rho,vs,fs,os))
  | VVar _ a <- v = a
  | otherwise     = error ""
lookType x r@(Env (Def _ decls rho,vs,fs,os)) = case lookup x decls of
  Just (a,_) -> eval r a
  Nothing -> lookType x (Env (rho,vs,fs,os))
lookType x (Env (Sub _ rho,vs,_:fs,os)) = lookType x (Env (rho,vs,fs,os))
lookType x (Env (Empty,_,_,_))          = error $ "lookType: not found " ++ show x

lookName :: C.Name -> Env -> C.Formula
lookName i (Env (Upd _ rho,v:vs,fs,os)) = lookName i (Env (rho,vs,fs,os))
lookName i (Env (Def _ _ rho,vs,fs,os)) = lookName i (Env (rho,vs,fs,os))
lookName i (Env (Sub j rho,vs,phi:fs,os)) | i == j    = phi
                                          | otherwise = lookName i (Env (rho,vs,fs,os))
lookName i _ = error $ "lookName: not found " ++ show i


  -----------------------------------------------------------------------
-- Nominal instances

instance C.Nominal Ctxt where
  support _ = []
  act e _   = e
  swap e _  = e

instance C.Nominal Env where
  support (Env (rho,vs,fs,os)) = C.support (rho,vs,fs,os)
  act (Env (rho,vs,fs,os)) iphi = Env $ C.act (rho,vs,fs,os) iphi
  swap (Env (rho,vs,fs,os)) ij = Env $ C.swap (rho,vs,fs,os) ij

instance C.Nominal Val where
  support v = case v of
    VU                      -> []
    Ter _ e                 -> C.support e
    VPi u v                 -> C.support [u,v]
    VComp a u ts            -> C.support (a,u,ts)
    VPathP a v0 v1          -> C.support [a,v0,v1]
    VPLam i v               -> i `delete` C.support v
    VSigma u v              -> C.support (u,v)
    VPair u v               -> C.support (u,v)
    VFst u                  -> C.support u
    VSnd u                  -> C.support u
    VCon _ vs               -> C.support vs
    VPCon _ a vs phis       -> C.support (a,vs,phis)
    VHComp a u ts           -> C.support (a,u,ts)
    VVar _ v                -> C.support v
    VOpaque _ v             -> C.support v
    VApp u v                -> C.support (u,v)
    VLam _ u v              -> C.support (u,v)
    VAppFormula u phi       -> C.support (u,phi)
    VSplit u v              -> C.support (u,v)
    VGlue a ts              -> C.support (a,ts)
    VGlueElem a ts          -> C.support (a,ts)
    VUnGlueElem a ts        -> C.support (a,ts)
    VCompU a ts             -> C.support (a,ts)
    VUnGlueElemU a b es     -> C.support (a,b,es)
    VIdPair u us            -> C.support (u,us)
    VId a u v               -> C.support (a,u,v)
    VIdJ a u c d x p        -> C.support [a,u,c,d,x,p]

```

```
act u (i, phi) | i `notElem` C.support u = u
               | otherwise =
  let acti :: C.Nominal a => a -> a
      acti u = C.act u (i, phi)
      sphi = C.support phi
  in case u of
       VU           -> VU
       Ter t e      -> Ter t (acti e)
       VPi a f      -> VPi (acti a) (acti f)
       VComp a v ts -> compLine (acti a) (acti v) (acti ts)
       VPathP a u v -> VPathP (acti a) (acti u) (acti v)
       VPLam j v | j == i -> u
                 | j `notElem` sphi -> VPLam j (acti v)
                 | otherwise -> VPLam k (acti (v `C.swap` (j,k)))
            where k = C.fresh (v,C.Atom i,phi)
       VSigma a f              -> VSigma (acti a) (acti f)
       VPair u v               -> VPair (acti u) (acti v)
       VFst u                  -> fstVal (acti u)
       VSnd u                  -> sndVal (acti u)
       VCon c vs               -> VCon c (acti vs)
       VPCon c a vs phis       -> pcon c (acti a) (acti vs) (acti phis)
       VHComp a u us           -> hComp (acti a) (acti u) (acti us)
       VVar x v                -> VVar x (acti v)
       VOpaque x v             -> VOpaque x (acti v)
       VAppFormula u psi       -> acti u @@ acti psi
       VApp u v                -> app (acti u) (acti v)
       VLam x t u              -> VLam x (acti t) (acti u)
       VSplit u v              -> app (acti u) (acti v)
       VGlue a ts              -> glue (acti a) (acti ts)
       VGlueElem a ts          -> glueElem (acti a) (acti ts)
       VUnGlueElem a ts        -> unglueElem (acti a) (acti ts)
       VUnGlueElemU a b es     -> unGlueU (acti a) (acti b) (acti es)
       VCompU a ts             -> compUniv (acti a) (acti ts)
       VIdPair u us            -> VIdPair (acti u) (acti us)
       VId a u v               -> VId (acti a) (acti u) (acti v)
       VIdJ a u c d x p        ->
         idJ (acti a) (acti u) (acti c) (acti d) (acti x) (acti p)

-- This increases efficiency as it won't trigger computation.
swap u ij@(i,j) =
  let sw :: C.Nominal a => a -> a
      sw u = C.swap u ij
  in case u of
       VU                      -> VU
       Ter t e                 -> Ter t (sw e)
       VPi a f                 -> VPi (sw a) (sw f)
       VComp a v ts            -> VComp (sw a) (sw v) (sw ts)
       VPathP a u v            -> VPathP (sw a) (sw u) (sw v)
       VPLam k v               -> VPLam (C.swapName k ij) (sw v)
       VSigma a f              -> VSigma (sw a) (sw f)
       VPair u v               -> VPair (sw u) (sw v)
       VFst u                  -> VFst (sw u)
       VSnd u                  -> VSnd (sw u)
       VCon c vs               -> VCon c (sw vs)
       VPCon c a vs phis       -> VPCon c (sw a) (sw vs) (sw phis)
       VHComp a u us           -> VHComp (sw a) (sw u) (sw us)
       VVar x v                -> VVar x (sw v)
       VOpaque x v             -> VOpaque x (sw v)
       VAppFormula u psi       -> VAppFormula (sw u) (sw psi)
       VApp u v                -> VApp (sw u) (sw v)
       VLam x u v              -> VLam x (sw u) (sw v)
       VSplit u v              -> VSplit (sw u) (sw v)
       VGlue a ts              -> VGlue (sw a) (sw ts)
       VGlueElem a ts          -> VGlueElem (sw a) (sw ts)
       VUnGlueElem a ts        -> VUnGlueElem (sw a) (sw ts)
       VUnGlueElemU a b es     -> VUnGlueElemU (sw a) (sw b) (sw es)
       VCompU a ts             -> VCompU (sw a) (sw ts)
       VIdPair u us            -> VIdPair (sw u) (sw us)
       VId a u v               -> VId (sw a) (sw u) (sw v)
       VIdJ a u c d x p        ->
```

```haskell
           VIdJ (sw a) (sw u) (sw c) (sw d) (sw x) (sw p)

-----------------------------------------------------------------------
-- The evaluator

eval :: Env -> Ter -> Val
eval rho@(Env (_,_,_,C.Nameless os)) v = case v of
  U                   -> VU
  App r s             -> app (eval rho r) (eval rho s)
  Var i
    | i `Set.member` os -> VOpaque i (lookType i rho)
    | otherwise       -> look i rho
  Pi t@(Lam _ a _)    -> VPi (eval rho a) (eval rho t)
  Sigma t@(Lam _ a _) -> VSigma (eval rho a) (eval rho t)
  Pair a b            -> VPair (eval rho a) (eval rho b)
  Fst a               -> fstVal (eval rho a)
  Snd a               -> sndVal (eval rho a)
  Where t decls       -> eval (defWhere decls rho) t
  Con name ts         -> VCon name (map (eval rho) ts)
  PCon name a ts phis ->
    pcon name (eval rho a) (map (eval rho) ts) (map (evalFormula rho) phis)
  Lam{}               -> Ter v rho
  Split{}             -> Ter v rho
  Sum{}               -> Ter v rho
  HSum{}              -> Ter v rho
  Undef{}             -> Ter v rho
  Hole{}              -> Ter v rho
  PathP a e0 e1       -> VPathP (eval rho a) (eval rho e0) (eval rho e1)
  PLam i t            -> let j = C.fresh rho
                         in VPLam j (eval (sub (i,C.Atom j) rho) t)
  AppFormula e phi    -> eval rho e @@ evalFormula rho phi
  Comp a t0 ts        ->
    compLine (eval rho a) (eval rho t0) (evalSystem rho ts)
  HComp a t0 ts       ->
    hComp (eval rho a) (eval rho t0) (evalSystem rho ts)
  Fill a t0 ts        ->
    fillLine (eval rho a) (eval rho t0) (evalSystem rho ts)
  Glue a ts           -> glue (eval rho a) (evalSystem rho ts)
  GlueElem a ts       -> glueElem (eval rho a) (evalSystem rho ts)
  UnGlueElem a ts     -> unglueElem (eval rho a) (evalSystem rho ts)
  Id a r s            -> VId (eval rho a) (eval rho r) (eval rho s)
  IdPair b ts         -> VIdPair (eval rho b) (evalSystem rho ts)
  IdJ a t c d x p     -> idJ (eval rho a) (eval rho t) (eval rho c)
                             (eval rho d) (eval rho x) (eval rho p)
  _                   -> error $ "Cannot evaluate " ++ show v

evals :: Env -> [(Ident,Ter)] -> [(Ident,Val)]
evals env bts = [ (b,eval env t) | (b,t) <- bts ]

evalFormula :: Env -> C.Formula -> C.Formula
evalFormula rho phi = case phi of
  C.Atom i         -> lookName i rho
  C.NegAtom i      -> C.negFormula (lookName i rho)
  phi1 C.:/\: phi2 -> evalFormula rho phi1 `C.andFormula` evalFormula rho phi2
  phi1 C.:\/: phi2 -> evalFormula rho phi1 `C.orFormula` evalFormula rho phi2
  _              -> phi

evalSystem :: Env -> C.System Ter -> C.System Val
evalSystem rho ts =
  let out = concat [ let betas = C.meetss [ C.invFormula (lookName i rho) d
                                          | (i,d) <- assocs alpha ]
                     in [ (beta,eval (rho `C.face` beta) talpha) | beta <- betas ]
                   | (alpha,talpha) <- assocs ts ]
  in C.mkSystem out

app :: Val -> Val -> Val
app u v = case (u,v) of
  (Ter (Lam x _ t) e,_)               -> eval (upd (x,v) e) t
  (Ter (Split _ _ _ nvs) e,VCon c vs) -> case lookupBranch c nvs of
    Just (OBranch _ xs t) -> eval (upds (zip xs vs) e) t
```

```
    _     -> error $ "app: missing case in split for " ++ c
  (Ter (Split _ _ _ nvs) e,VPCon c _ us phis) -> case lookupBranch c nvs of
    Just (PBranch _ xs is t) -> eval (subs (zip is phis) (upds (zip xs us) e)) t
    _ -> error $ "app: missing case in split for " ++ c
  (Ter (Split _ _ ty hbr) e,VHComp a w ws) -> case eval e ty of
    VPi _ f -> let j   = C.fresh (e,v)
                   wsj = Map.map (@@ j) ws
                   w'  = app u w
                   ws' = mapWithKey (\alpha -> app (u `C.face` alpha)) wsj
                   -- a should be constant
               in comp j (app f (fill j a w wsj)) w' ws'
    _ -> error $ "app: Split annotation not a Pi type " ++ show u
  (Ter Split{} _,_) | isNeutral v         -> VSplit u v
  (VComp (VPLam i (VPi a f)) li0 ts,vi1) ->
    let j       = C.fresh (u,vi1)
        (aj,fj) = (a,f) `C.swap` (i,j)
        tsj     = Map.map (@@ j) ts
        v       = transFillNeg j aj vi1
        vi0     = transNeg j aj vi1
    in comp j (app fj v) (app li0 vi0)
              (intersectionWith app tsj (C.border v tsj))
  _ | isNeutral u       -> VApp u v
  _                     -> error $ "app \n  " ++ show u ++ "\n  " ++ show v

fstVal, sndVal :: Val -> Val
fstVal (VPair a b)     = a
fstVal u | isNeutral u = VFst u
fstVal u               = error $ "fstVal: " ++ show u ++ " is not neutral."
sndVal (VPair a b)     = b
sndVal u | isNeutral u = VSnd u
sndVal u               = error $ "sndVal: " ++ show u ++ " is not neutral."

-- infer the type of a neutral value
inferType :: Val -> Val
inferType v = case v of
  VVar _ t -> t
  VOpaque _ t -> t
  Ter (Undef _ t) rho -> eval rho t
  VFst t -> case inferType t of
    VSigma a _ -> a
    ty         -> error $ "inferType: expected Sigma type for " ++ show v
                  ++ ", got " ++ show ty
  VSnd t -> case inferType t of
    VSigma _ f -> app f (VFst t)
    ty         -> error $ "inferType: expected Sigma type for " ++ show v
                  ++ ", got " ++ show ty
  VSplit s@(Ter (Split _ _ t _) rho) v1 -> case eval rho t of
    VPi _ f -> app f v1
    ty      -> error $ "inferType: Pi type expected for split annotation in "
               ++ show v ++ ", got " ++ show ty
  VApp t0 t1 -> case inferType t0 of
    VPi _ f -> app f t1
    ty      -> error $ "inferType: expected Pi type for " ++ show v
               ++ ", got " ++ show ty
  VAppFormula t phi -> case inferType t of
    VPathP a _ _ -> a @@ phi
    ty         -> error $ "inferType: expected PathP type for " ++ show v
                  ++ ", got " ++ show ty
  VComp a _ _ -> a @@ C.One
--  VUnGlueElem _ b _  -> b   -- This is wrong! Store the type??
  VUnGlueElemU _ b _ -> b
  VIdJ _ _ c _ x p -> app (app c x) p
  _ -> error $ "inferType: not neutral " ++ show v

(@@) :: C.ToFormula a => Val -> a -> Val
(VPLam i u) @@ phi         = u `C.act` (i,C.toFormula phi)
v@(Ter Hole{} _) @@ phi    = VAppFormula v (C.toFormula phi)
v @@ phi | isNeutral v     = case (inferType v,C.toFormula phi) of
  (VPathP _ a0 _,C.Dir 0) -> a0
  (VPathP _ _ a1,C.Dir 1) -> a1
```

```haskell
  _                      -> VAppFormula v (C.toFormula phi)
v @@ phi                     = error $ "(@@): " ++ show v ++ " should be neutral."

-- Applying a *C.fresh* name.
(@@@) :: Val -> C.Name -> Val
(VPLam i u) @@@ j = u `C.swap` (i,j)
v @@@ j           = VAppFormula v (C.toFormula j)


-------------------------------------------------------------------------------
-- Composition and filling

comp :: C.Name -> Val -> Val -> C.System Val -> Val
comp i a u ts | C.eps `member` ts = (ts ! C.eps) `C.face` (i C.~> 1)
comp i a u ts = case a of
  VPathP p v0 v1 -> let j = C.fresh (C.Atom i,a,u,ts)
                    in VPLam j $ comp i (p @@ j) (u @@ j) $
                         C.insertsSystem [(j C.~> 0,v0),(j C.~> 1,v1)] (Map.map (@@ j) ts)
  VId b v0 v1 -> case u of
    VIdPair r _ | all isIdPair (elems ts) ->
      let j = C.fresh (C.Atom i,a,u,ts)
          VIdPair z _ @@@ phi = z @@ phi
          sys (VIdPair _ ws)  = ws
          w = VPLam j $ comp i b (r @@ j) $
                          C.insertsSystem [(j C.~> 0,v0),(j C.~> 1,v1)]
                            (Map.map (@@@ j) ts)
      in VIdPair w (C.joinSystem (Map.map sys (ts `C.face` (i C.~> 1))))
    _ -> VComp (VPLam i a) u (Map.map (VPLam i) ts)
  VSigma a f -> VPair ui1 comp_u2
    where (t1s, t2s) = (Map.map fstVal ts, Map.map sndVal ts)
          (u1,  u2)  = (fstVal u, sndVal u)
          fill_u1    = fill i a u1 t1s
          ui1        = comp i a u1 t1s
          comp_u2    = comp i (app f fill_u1) u2 t2s
  VPi{} -> VComp (VPLam i a) u (Map.map (VPLam i) ts)
  VU -> compUniv u (Map.map (VPLam i) ts)
  VCompU a es | not (isNeutralU i es u ts)  -> compU i a es u ts
  VGlue b equivs | not (isNeutralGlue i equivs u ts) -> compGlue i b equivs u ts
  Ter (Sum _ _ nass) env -> case u of
    VCon n us | all isCon (elems ts) -> case lookupLabel n nass of
      Just as -> let tsus = C.transposeSystemAndList (Map.map unCon ts) us
                 in VCon n $ comps i as env tsus
      Nothing -> error $ "comp: missing constructor in labelled sum " ++ n
    _ -> VComp (VPLam i a) u (Map.map (VPLam i) ts)
  Ter (HSum _ _ nass) env -> compHIT i a u ts
  _ -> VComp (VPLam i a) u (Map.map (VPLam i) ts)

compNeg :: C.Name -> Val -> Val -> C.System Val -> Val
compNeg i a u ts = comp i (a `C.sym` i) u (ts `C.sym` i)

compLine :: Val -> Val -> C.System Val -> Val
compLine a u ts = comp i (a @@ i) u (Map.map (@@ i) ts)
  where i = C.fresh (a,u,ts)

compConstLine :: Val -> Val -> C.System Val -> Val
compConstLine a u ts = comp i a u (Map.map (@@ i) ts)
  where i = C.fresh (a,u,ts)

comps :: C.Name -> [(Ident,Ter)] -> Env -> [(C.System Val,Val)] -> [Val]
comps i []         _ []         = []
comps i ((x,a):as) e ((ts,u):tsus) =
  let v   = fill i (eval e a) u ts
      vi1 = comp i (eval e a) u ts
      vs  = comps i as (upd (x,v) e) tsus
  in vi1 : vs
comps _ _ _ _ = error "comps: different lengths of types and values"

fill :: C.Name -> Val -> Val -> C.System Val -> Val
fill i a u ts =
```

```haskell
  comp j (a `C.conj` (i,j)) u (C.insertSystem (i C.~> 0) u (ts `C.conj` (i,j)))
  where j = C.fresh (C.Atom i,a,u,ts)

fillNeg :: C.Name -> Val -> Val -> C.System Val -> Val
fillNeg i a u ts = (fill i (a `C.sym` i) u (ts `C.sym` i)) `C.sym` i

fillLine :: Val -> Val -> C.System Val -> Val
fillLine a u ts = VPLam i $ fill i (a @@ i) u (Map.map (@@ i) ts)
  where i = C.fresh (a,u,ts)

-- fills :: C.Name -> [(Ident,Ter)] -> Env -> [(C.System Val,Val)] -> [Val]
-- fills i []         _ []         = []
-- fills i ((x,a):as) e ((ts,u):tsus) =
--   let v  = fill i (eval e a) ts u
--       vs = fills i as (Upd e (x,v)) tsus
--   in v : vs
-- fills _ _ _ _ = error "fills: different lengths of types and values"


-----------------------------------------------------------
-- Transport and squeeze (defined using comp)

trans :: C.Name -> Val -> Val -> Val
trans i v0 v1 = comp i v0 v1 empty

transNeg :: C.Name -> Val -> Val -> Val
transNeg i a u = trans i (a `C.sym` i) u

transLine :: Val -> Val -> Val
transLine u v = trans i (u @@ i) v
  where i = C.fresh (u,v)

transNegLine :: Val -> Val -> Val
transNegLine u v = transNeg i (u @@ i) v
  where i = C.fresh (u,v)

-- TODO: define in terms of comps?
transps :: C.Name -> [(Ident,Ter)] -> Env -> [Val] -> [Val]
transps i []         _ []     = []
transps i ((x,a):as) e (u:us) =
  let v   = transFill i (eval e a) u
      vi1 = trans i (eval e a) u
      vs  = transps i as (upd (x,v) e) us
  in vi1 : vs
transps _ _ _ _ = error "transps: different lengths of types and values"

transFill :: C.Name -> Val -> Val -> Val
transFill i a u = fill i a u empty

transFillNeg :: C.Name -> Val -> Val -> Val
transFillNeg i a u = (transFill i (a `C.sym` i) u) `C.sym` i

-- Given u of type a "squeeze i a u" connects in the direction i
-- trans i a u(i=0) to u(i=1)
squeeze :: C.Name -> Val -> Val -> Val
squeeze i a u = comp j (a `C.disj` (i,j)) u $ C.mkSystem [ (i C.~> 1, ui1) ]
  where j   = C.fresh (C.Atom i,a,u)
        ui1 = u `C.face` (i C.~> 1)

squeezes :: C.Name -> [(Ident,Ter)] -> Env -> [Val] -> [Val]
squeezes i xas e us = comps j xas (e `C.disj` (i,j)) us'
  where j   = C.fresh (us,e,C.Atom i)
        us' = [ (C.mkSystem [(i C.~> 1, u `C.face` (i C.~> 1))],u) | u <- us ]


-------------------------------------------------------------------------------
-- | Id

idJ :: Val -> Val -> Val -> Val -> Val -> Val -> Val
idJ a v c d x p = case p of
```

```haskell
  VIdPair w ws -> comp i (app (app c (w @@ i)) w') d
                    (C.border d (C.shape ws))
    where w' = VIdPair (VPLam j $ w @@ (C.Atom i C.:/\: C.Atom j))
                  (C.insertSystem (i C.~> 0) v ws)
          i:j:_ = C.freshs [a,v,c,d,x,p]
  _ -> VIdJ a v c d x p

isIdPair :: Val -> Bool
isIdPair VIdPair{} = True
isIdPair _         = False


-------------------------------------------------------------------------------
-- | HITs

pcon :: LIdent -> Val -> [Val] -> [C.Formula] -> Val
pcon c a@(Ter (HSum _ _ lbls) rho) us phis = case lookupPLabel c lbls of
  Just (tele,is,ts) | C.eps `member` vs -> vs ! C.eps
                    | otherwise       -> VPCon c a us phis
    where rho' = subs (zip is phis) (updsTele tele us rho)
          vs   = evalSystem rho' ts
  Nothing           -> error "pcon"
pcon c a us phi     = VPCon c a us phi

compHIT :: C.Name -> Val -> Val -> C.System Val -> Val
compHIT i a u us
  | isNeutral u || isNeutralSystem us =
      VComp (VPLam i a) u (Map.map (VPLam i) us)
  | otherwise =
      hComp (a `C.face` (i C.~> 1)) (transpHIT i a u) $
        mapWithKey (\alpha uAlpha ->
                     VPLam i $ squeezeHIT i (a `C.face` alpha) uAlpha) us

-- Given u of type a(i=0), transpHIT i a u is an element of a(i=1).
transpHIT :: C.Name -> Val -> Val -> Val
transpHIT i a@(Ter (HSum _ _ nass) env) u =
 let j = C.fresh (a,u)
     aij = C.swap a (i,j)
 in
 case u of
  VCon n us -> case lookupLabel n nass of
    Just as -> VCon n (transps i as env us)
    Nothing -> error $ "transpHIT: missing constructor in labelled sum " ++ n
  VPCon c _ ws0 phis -> case lookupLabel c nass of
    Just as -> pcon c (a `C.face` (i C.~> 1)) (transps i as env ws0) phis
    Nothing -> error $ "transpHIT: missing path constructor " ++ c
  VHComp _ v vs ->
    hComp (a `C.face` (i C.~> 1)) (transpHIT i a v) $
      mapWithKey (\alpha vAlpha ->
                   VPLam j $ transpHIT j (aij `C.face` alpha) (vAlpha @@ j)) vs
  _ -> error $ "transpHIT: neutral " ++ show u

-- given u(i) of type a(i) "squeezeHIT i a u" connects in the direction i
-- transHIT i a u(i=0) to u(i=1) in a(1)
squeezeHIT :: C.Name -> Val -> Val -> Val
squeezeHIT i a@(Ter (HSum _ _ nass) env) u =
 let j = C.fresh (a,u)
 in
 case u of
  VCon n us -> case lookupLabel n nass of
    Just as -> VCon n (squeezes i as env us)
    Nothing -> error $ "squeezeHIT: missing constructor in labelled sum " ++ n
  VPCon c _ ws0 phis -> case lookupLabel c nass of
    Just as -> pcon c (a `C.face` (i C.~> 1)) (squeezes i as env ws0) phis
    Nothing -> error $ "squeezeHIT: missing path constructor " ++ c
  VHComp _ v vs -> hComp (a `C.face` (i C.~> 1)) (squeezeHIT i a v) $
      mapWithKey
        (\alpha vAlpha -> case Map.lookup i alpha of
          Nothing   -> VPLam j $ squeezeHIT i (a `C.face` alpha) (vAlpha @@ j)
          Just C.Zero -> VPLam j $ transpHIT i
```

```haskell
                         (a `C.face` (Map.delete i alpha)) (vAlpha @@ j)
          Just C.One  -> vAlpha)
        vs
  _ -> error $ "squeezeHIT: neutral " ++ show u

hComp :: Val -> Val -> C.System Val -> Val
hComp a u us | C.eps `member` us = (us ! C.eps) @@ C.One
             | otherwise       = VHComp a u us

-------------------------------------------------------------------------------
-- | Glue

-- An equivalence for a type a is a triple (t,f,p) where
-- t : U
-- f : t -> a
-- p : (x : a) -> isContr ((y:t) * Id a x (f y))
-- with isContr c = (z : c) * ((z' : C) -> Id c z z')

-- Extraction functions for getting a, f, s and t:
equivDom :: Val -> Val
equivDom = fstVal

equivFun :: Val -> Val
equivFun = fstVal . sndVal

equivContr :: Val -> Val
equivContr =  sndVal . sndVal

glue :: Val -> C.System Val -> Val
glue b ts | C.eps `member` ts = equivDom (ts ! C.eps)
          | otherwise       = VGlue b ts

glueElem :: Val -> C.System Val -> Val
glueElem v us | C.eps `member` us = us ! C.eps
glueElem v us = VGlueElem v us

unglueElem :: Val -> C.System Val -> Val
unglueElem w isos | C.eps `member` isos = app (equivFun (isos ! C.eps)) w
                  | otherwise         = case w of
                                          VGlueElem v us -> v
                                          _ -> VUnGlueElem w isos

unGlue :: Val -> Val -> C.System Val -> Val
unGlue w b equivs | C.eps `member` equivs = app (equivFun (equivs ! C.eps)) w
                  | otherwise           = case w of
                                            VGlueElem v us -> v
                                            _ -> error ("unglue: neutral" ++ show w)

isNeutralGlue :: C.Name -> C.System Val -> Val -> C.System Val -> Bool
isNeutralGlue i equivs u0 ts = (C.eps `notMember` equivsi0 && isNeutral u0) ||
  any (\(alpha,talpha) ->
           C.eps `notMember` (equivs `C.face` alpha) && isNeutral talpha)
    (assocs ts)
  where equivsi0 = equivs `C.face` (i C.~> 0)

-- this is exactly the same as isNeutralGlue?
isNeutralU :: C.Name -> C.System Val -> Val -> C.System Val -> Bool
isNeutralU i eqs u0 ts = (C.eps `notMember` eqsi0 && isNeutral u0) ||
  any (\(alpha,talpha) ->
           C.eps `notMember` (eqs `C.face` alpha) && isNeutral talpha)
    (assocs ts)
  where eqsi0 = eqs `C.face` (i C.~> 0)

-- Extend the system ts to a total element in b given q : isContr b
extend :: Val -> Val -> C.System Val -> Val
extend b q ts = comp i b (fstVal q) ts'
  where i = C.fresh (b,q,ts)
        ts' = mapWithKey
                (\alpha tAlpha -> app ((sndVal q) `C.face` alpha) tAlpha @@ i) ts

```

```
-- psi/b corresponds to ws
-- b0    corresponds to wi0
-- a0    corresponds to vi0
-- psi/a corresponds to vs
-- a1'   corresponds to vi1'
-- equivs' corresponds to delta
-- ti1'  corresponds to usi1'
compGlue :: C.Name -> Val -> C.System Val -> Val -> C.System Val -> Val
compGlue i a equivs wi0 ws = glueElem vi1 usi1
  where ai1 = a `C.face` (i C.~> 1)
        vs  = mapWithKey
                (\alpha wAlpha ->
                  unGlue wAlpha (a `C.face` alpha) (equivs `C.face` alpha)) ws

        vsi1 = vs `C.face` (i C.~> 1) -- same as: C.border vi1 vs
        vi0  = unGlue wi0 (a `C.face` (i C.~> 0)) (equivs `C.face` (i C.~> 0)) -- in a(i0)

        vi1'  = comp i a vi0 vs           -- in a(i1)

        equivsI1 = equivs `C.face` (i C.~> 1)
        equivs'  = filterWithKey (\alpha _ -> i `notMember` alpha) equivs

        us'    = mapWithKey (\gamma equivG ->
                   fill i (equivDom equivG) (wi0 `C.face` gamma) (ws `C.face` gamma))
                 equivs'
        usi1'  = mapWithKey (\gamma equivG ->
                   comp i (equivDom equivG) (wi0 `C.face` gamma) (ws `C.face` gamma))
                 equivs'

        -- path in ai1 between vi1 and f(i1) usi1' on equivs'
        ls'    = mapWithKey (\gamma equivG ->
                   pathComp i (a `C.face` gamma) (vi0 `C.face` gamma)
                     (equivFun equivG `app` (us' ! gamma)) (vs `C.face` gamma))
                 equivs'

        fibersys = intersectionWith VPair usi1' ls' -- on equivs'

        wsi1 = ws `C.face` (i C.~> 1)
        fibersys' = mapWithKey
          (\gamma equivG ->
            let fibsgamma = intersectionWith (\ x y -> VPair x (constPath y))
                              (wsi1 `C.face` gamma) (vsi1 `C.face` gamma)
            in extend (mkFiberType (ai1 `C.face` gamma) (vi1' `C.face` gamma) equivG)
                 (app (equivContr equivG) (vi1' `C.face` gamma))
                 (fibsgamma `C.unionSystem` (fibersys `C.face` gamma))) equivsI1

        vi1 = compConstLine ai1 vi1'
                (Map.map sndVal fibersys' `C.unionSystem` Map.map constPath vsi1)

        usi1 = Map.map fstVal fibersys'

mkFiberType :: Val -> Val -> Val -> Val
mkFiberType a x equiv = eval rho $
  Sigma $ Lam "y" tt (PathP (PLam (C.Name "_") ta) tx (App tf ty))
  where [ta,tx,ty,tf,tt] = map Var ["a","x","y","f","t"]
        rho = upds [("a",a),("x",x),("f",equivFun equiv)
                   ,("t",equivDom equiv)] emptyEnv

-- Assumes u' : A is a solution of us + (i0 -> u0)
-- The output is an L-path in A(i1) between comp i u0 us and u'(i1)
pathComp :: C.Name -> Val -> Val -> Val -> C.System Val -> Val
pathComp i a u0 u' us = VPLam j $ comp i a u0 us'
  where j   = C.fresh (C.Atom i,a,us,u0,u')
        us' = C.insertsSystem [(j C.~> 1, u')] us

-------------------------------------------------------------------------------
-- | Composition in the Universe

-- any path between types define an equivalence
```

```
eqFun :: Val -> Val -> Val
eqFun = transNegLine

unGlueU :: Val -> Val -> C.System Val -> Val
unGlueU w b es | C.eps `Map.member` es = eqFun (es ! C.eps) w
               | otherwise          = case w of
                                       VGlueElem v us   -> v
                                       _ -> VUnGlueElemU w b es

compUniv :: Val -> C.System Val -> Val
compUniv b es | C.eps `Map.member` es = (es ! C.eps) @@ C.One
              | otherwise          = VCompU b es

compU :: C.Name -> Val -> C.System Val -> Val -> C.System Val -> Val
compU i a eqs wi0 ws = glueElem vi1 usi1
  where ai1 = a `C.face` (i C.~> 1)
        vs  = mapWithKey
                (\alpha wAlpha ->
                  unGlueU wAlpha (a `C.face` alpha) (eqs `C.face` alpha)) ws

        vsi1 = vs `C.face` (i C.~> 1) -- same as: C.border vi1 vs
        vi0  = unGlueU wi0 (a `C.face` (i C.~> 0)) (eqs `C.face` (i C.~> 0)) -- in a(i0)

        vi1'  = comp i a vi0 vs           -- in a(i1)

        eqsI1 = eqs `C.face` (i C.~> 1)
        eqs'  = filterWithKey (\alpha _ -> i `notMember` alpha) eqs

        us'    = mapWithKey (\gamma eqG ->
                   fill i (eqG @@ C.One) (wi0 `C.face` gamma) (ws `C.face` gamma))
                 eqs'
        usi1'  = mapWithKey (\gamma eqG ->
                   comp i (eqG @@ C.One) (wi0 `C.face` gamma) (ws `C.face` gamma))
                 eqs'

        -- path in ai1 between vi1 and f(i1) usi1' on eqs'
        ls'    = mapWithKey (\gamma eqG ->
                   pathComp i (a `C.face` gamma) (vi0 `C.face` gamma)
                     (eqFun eqG (us' ! gamma)) (vs `C.face` gamma))
                 eqs'

        fibersys = intersectionWith (\ x y -> (x,y)) usi1' ls' -- on eqs'

        wsi1 = ws `C.face` (i C.~> 1)
        fibersys' = mapWithKey
          (\gamma eqG ->
            let fibsgamma = intersectionWith (\ x y -> (x,constPath y))
                              (wsi1 `C.face` gamma) (vsi1 `C.face` gamma)
            in lemEq eqG (vi1' `C.face` gamma)
                     (fibsgamma `C.unionSystem` (fibersys `C.face` gamma))) eqsI1

        vi1 = compConstLine ai1 vi1'
                (Map.map snd fibersys' `C.unionSystem` Map.map constPath vsi1)

        usi1 = Map.map fst fibersys'

lemEq :: Val -> Val -> C.System (Val,Val) -> (Val,Val)
lemEq eq b aps = (a,VPLam i (compNeg j (eq @@ j) p1 thetas'))
 where
   i:j:_ = C.freshs (eq,b,aps)
   ta = eq @@ C.One
   p1s = mapWithKey (\alpha (aa,pa) ->
              let eqaj = (eq `C.face` alpha) @@ j
                  ba = b `C.face` alpha
              in comp j eqaj (pa @@ i)
                   (C.mkSystem [ (i C.~>0,transFill j eqaj ba)
                             , (i C.~>1,transFillNeg j eqaj aa)])) aps
   thetas = mapWithKey (\alpha (aa,pa) ->
              let eqaj = (eq `C.face` alpha) @@ j
                  ba = b `C.face` alpha
```

```
            in fill j eqaj (pa @@ i)
                 (C.mkSystem [ (i C.~>0,transFill j eqaj ba)
                             , (i C.~>1,transFillNeg j eqaj aa)])) aps

    a  = comp i ta (trans i (eq @@ i) b) p1s
    p1 = fill i ta (trans i (eq @@ i) b) p1s

    thetas' = C.insertsSystem [ (i C.~> 0,transFill j (eq @@ j) b)
                            , (i C.~> 1,transFillNeg j (eq @@ j) a)] thetas

-- Old version:
-- This version triggers the following error when checking the normal form of corrUniv:
-- Parsed "examples/nunivalence2.ctt" successfully!
-- Resolver failed: Cannot resolve name !3 at position (7,30062) in module nunivalence2
-- compU :: Name -> Val -> C.System Val -> Val -> C.System Val -> Val
-- compU i b es wi0 ws = glueElem vi1'' usi1''
--   where bi1 = b `C.face` (i C.~> 1)
--         vs   = mapWithKey (\alpha wAlpha ->
--                  unGlueU wAlpha (b `C.face` alpha) (es `C.face` alpha)) ws
--         vsi1 = vs `C.face` (i C.~> 1) -- same as: C.border vi1 vs
--         vi0  = unGlueU wi0 (b `C.face` (i C.~> 0)) (es `C.face` (i C.~> 0)) -- in b(i0)

--         v    = fill i b vi0 vs           -- in b
--         vi1  = comp i b vi0 vs           -- is v `C.face` (i C.~> 1) in b(i1)

--         esI1 = es `C.face` (i C.~> 1)
--         es'  = filterWithKey (\alpha _ -> i `Map.notMember` alpha) es
--         es'' = filterWithKey (\alpha _ -> alpha `Map.notMember` es) esI1

--         us'    = mapWithKey (\gamma eGamma ->
--                    fill i (eGamma @@ C.One) (wi0 `C.face` gamma) (ws `C.face` gamma))
--                  es'
--         usi1'  = mapWithKey (\gamma eGamma ->
--                    comp i (eGamma @@ C.One) (wi0 `C.face` gamma) (ws `C.face` gamma))
--                  es'

--         ls'    = mapWithKey (\gamma eGamma ->
--                      pathComp i (b `C.face` gamma) (v `C.face` gamma)
--                        (transNegLine eGamma (us' ! gamma)) (vs `C.face` gamma))
--                    es'

--         vi1' = compLine (constPath bi1) vi1
--                  (ls' `C.unionSystem` Map.map constPath vsi1)

--         wsi1 = ws `C.face` (i C.~> 1)

--         -- for gamma in es'', (i1) gamma is in es, so wsi1 gamma
--         -- is in the domain of isoGamma
--         uls'' = mapWithKey (\gamma eGamma ->
--                   isoToEquivU (bi1 `C.face` gamma) eGamma
--                     ((usi1' `C.face` gamma) `C.unionSystem` (wsi1 `C.face` gamma))
--                     (vi1' `C.face` gamma))
--                   es''

--         vsi1' = Map.map constPath $ C.border vi1' es' `C.unionSystem` vsi1

--         vi1'' = compLine (constPath bi1) vi1'
--                   (Map.map snd uls'' `C.unionSystem` vsi1')

--         usi1'' = Map.mapWithKey (\gamma _ ->
--                      if gamma `Map.member` usi1' then usi1' ! gamma
--                      else fst (uls'' ! gamma))
--                   esI1

-- IsoToEquiv, takes a line eq in U, a system us and a value v, s.t. f us =
-- C.border v. Outputs (u,p) s.t. C.border u = us and a path p between v
-- and f u, where f is transNegLine eq
-- isoToEquivU :: Val -> Val -> C.System Val -> Val -> (Val, Val)
-- isoToEquivU b eq us v = (u, VPLam i theta)
```

```haskell
--   where i:j:_   = C.freshs (b,eq,us,v)
--         ej      = eq @@ j
--         a       = eq @@ C.One
--         ws      = mapWithKey (\alpha uAlpha ->
--                                   transFillNeg j (ej `C.face` alpha) uAlpha) us
--         u       = comp j ej v ws
--         w       = fill j ej v ws
--         xs      = C.insertSystem (i C.~> 0) w $
--                     C.insertSystem (i C.~> 1) (transFillNeg j ej u) $ ws
--         theta   = compNeg j ej u xs

-- Old version:
-- isoToEquivU :: Val -> Val -> System Val -> Val -> (Val, Val)
-- isoToEquivU b eq us v = (u, VPLam i theta'')
--   where i:j:_   = C.freshs (b,eq,us,v)
--         a       = eq @@ C.One
--         g       = transLine
--         f       = transNegLine
--         s e y   = VPLam j $ compNeg i (e @@ i) (trans i (e @@ i) y)
--                     (C.mkSystem [(j C.~> 0, transFill j (e @@ j) y)
--                                 ,(j C.~> 1, transFillNeg j (e @@ j)
--                                               (trans j (e @@ j) y))])
--         t e x   = VPLam j $ comp i (e @@ i) (transNeg i (e @@ i) x)
--                     (C.mkSystem [(j C.~> 0, transFill j (e @@ j)
--                                               (transNeg j (e @@ j) x))
--                                 ,(j C.~> 1, transFillNeg j (e @@ j) x)])
--         gv      = g eq v
--         us'     = mapWithKey (\alpha uAlpha ->
--                                   t (eq `C.face` alpha) uAlpha @@ i) us
--         theta   = fill i a gv us'
--         u       = comp i a gv us'  -- Same as "theta `C.face` (i C.~> 1)"
--         ws      = C.insertSystem (i C.~> 0) gv $
--                     C.insertSystem (i C.~> 1) (t eq u @@ j) $
--                     mapWithKey
--                       (\alpha uAlpha ->
--                         t (eq `C.face` alpha) uAlpha @@ (C.Atom i :/\: C.Atom j)) us
--         theta'  = compNeg j a theta ws
--         xs      = C.insertSystem (i C.~> 0) (s eq v @@ j) $
--                     C.insertSystem (i C.~> 1) (s eq (f eq u) @@ j) $
--                     mapWithKey
--                       (\alpha uAlpha ->
--                          s (eq `C.face` alpha) (f (eq `C.face` alpha) uAlpha) @@ j) us
--         theta'' = comp j b (f eq theta') xs


-------------------------------------------------------------------------------
-- | Conversion

class Convertible a where
  conv :: [String] -> a -> a -> Bool

isCompSystem :: (C.Nominal a, Convertible a) => [String] -> C.System a -> Bool
isCompSystem ns ts = and [ conv ns (getFace alpha beta) (getFace beta alpha)
                         | (alpha,beta) <- C.allCompatible (keys ts) ]
    where getFace a b = C.face (ts ! a) (b `C.minus` a)

instance Convertible Env where
  conv ns (Env (rho1,vs1,fs1,os1)) (Env (rho2,vs2,fs2,os2)) =
      conv ns (rho1,vs1,fs1,os1) (rho2,vs2,fs2,os2)

instance Convertible Val where
  conv ns u v | u == v    = True
              | otherwise =
    let j = C.fresh (u,v)
    in case (u,v) of
      (Ter (Lam x a u) e,Ter (Lam x' a' u') e') ->
        let v@(VVar n _) = mkVarNice ns x (eval e a)
        in conv (n:ns) (eval (upd (x,v) e) u) (eval (upd (x',v) e') u')
      (Ter (Lam x a u) e,u') ->
        let v@(VVar n _) = mkVarNice ns x (eval e a)
```

```
        in conv (n:ns) (eval (upd (x,v) e) u) (app u' v)
      (u',Ter (Lam x a u) e) ->
        let v@(VVar n _) = mkVarNice ns x (eval e a)
        in conv (n:ns) (app u' v) (eval (upd (x,v) e) u)
      (Ter (Split _ p _ _) e,Ter (Split _ p' _ _) e') -> (p == p') && conv ns e e'
      (Ter (Sum p _ _) e,Ter (Sum p' _ _) e')         -> (p == p') && conv ns e e'
      (Ter (HSum p _ _) e,Ter (HSum p' _ _) e')       -> (p == p') && conv ns e e'
      (Ter (Undef p _) e,Ter (Undef p' _) e') -> p == p' && conv ns e e'
      (Ter (Hole p) e,Ter (Hole p') e') -> p == p' && conv ns e e'
      -- (Ter Hole{} e,_) -> True
      -- (_,Ter Hole{} e') -> True
      (VPi u v,VPi u' v') ->
        let w@(VVar n _) = mkVarNice ns "X" u
        in conv ns u u' && conv (n:ns) (app v w) (app v' w)
      (VSigma u v,VSigma u' v') ->
        let w@(VVar n _) = mkVarNice ns "X" u
        in conv ns u u' && conv (n:ns) (app v w) (app v' w)
      (VCon c us,VCon c' us')   -> (c == c') && conv ns us us'
      (VPCon c v us phis,VPCon c' v' us' phis') ->
        (c == c') && conv ns (v,us,phis) (v',us',phis')
      (VPair u v,VPair u' v')    -> conv ns u u' && conv ns v v'
      (VPair u v,w)              -> conv ns u (fstVal w) && conv ns v (sndVal w)
      (w,VPair u v)              -> conv ns (fstVal w) u && conv ns (sndVal w) v
      (VFst u,VFst u')           -> conv ns u u'
      (VSnd u,VSnd u')           -> conv ns u u'
      (VApp u v,VApp u' v')      -> conv ns u u' && conv ns v v'
      (VSplit u v,VSplit u' v')  -> conv ns u u' && conv ns v v'
      (VOpaque x _, VOpaque x' _) -> x == x'
      (VVar x _, VVar x' _)       -> x == x'
      (VPathP a b c,VPathP a' b' c') -> conv ns a a' && conv ns b b' && conv ns c c'
      (VPLam i a,VPLam i' a')    -> conv ns (a `C.swap` (i,j)) (a' `C.swap` (i',j))
      (VPLam i a,p')             -> conv ns (a `C.swap` (i,j)) (p' @@ j)
      (p,VPLam i' a')            -> conv ns (p @@ j) (a' `C.swap` (i',j))
      (VAppFormula u x,VAppFormula u' x') -> conv ns (u,x) (u',x')
      (VComp a u ts,VComp a' u' ts')      -> conv ns (a,u,ts) (a',u',ts')
      (VHComp a u ts,VHComp a' u' ts')    -> conv ns (a,u,ts) (a',u',ts')
      (VGlue v equivs,VGlue v' equivs')   -> conv ns (v,equivs) (v',equivs')
      (VGlueElem (VUnGlueElem b equivs) ts,g) -> conv ns (C.border b equivs,b) (ts,g)
      (g,VGlueElem (VUnGlueElem b equivs) ts) -> conv ns (C.border b equivs,b) (ts,g)
      (VGlueElem (VUnGlueElemU b _ equivs) ts,g) -> conv ns (C.border b equivs,b) (ts,g)
      (g,VGlueElem (VUnGlueElemU b _ equivs) ts) -> conv ns (C.border b equivs,b) (ts,g)
      (VGlueElem u us,VGlueElem u' us')   -> conv ns (u,us) (u',us')
      (VUnGlueElemU u _ _,VUnGlueElemU u' _ _) -> conv ns u u'
      (VUnGlueElem u _,VUnGlueElem u' _) -> conv ns u u'
      (VCompU u es,VCompU u' es')              -> conv ns (u,es) (u',es')
      (VIdPair v vs,VIdPair v' vs')          -> conv ns (v,vs) (v',vs')
      (VId a u v,VId a' u' v')               -> conv ns (a,u,v) (a',u',v')
      (VIdJ a u c d x p,VIdJ a' u' c' d' x' p') ->
        conv ns [a,u,c,d,x,p] [a',u',c',d',x',p']
      _                                      -> False

instance Convertible Ctxt where
  conv _ _ _ = True

instance Convertible () where
  conv _ _ _ = True

instance (Convertible a, Convertible b) => Convertible (a, b) where
  conv ns (u, v) (u', v') = conv ns u u' && conv ns v v'

instance (Convertible a, Convertible b, Convertible c)
      => Convertible (a, b, c) where
  conv ns (u, v, w) (u', v', w') = conv ns (u,(v,w)) (u',(v',w'))

instance (Convertible a,Convertible b,Convertible c,Convertible d)
      => Convertible (a,b,c,d) where
  conv ns (u,v,w,x) (u',v',w',x') = conv ns (u,v,(w,x)) (u',v',(w',x'))

instance Convertible a => Convertible [a] where
  conv ns us us' = length us == length us' &&
```

```haskell
                  and [conv ns u u' | (u,u') <- zip us us']

instance Convertible a => Convertible (C.System a) where
  conv ns ts ts' = keys ts == keys ts' &&
                   and (elems (intersectionWith (conv ns) ts ts'))

instance Convertible C.Formula where
  conv _ phi psi = C.dnf phi == C.dnf psi

instance Convertible (C.Nameless a) where
  conv _ _ _ = True

-------------------------------------------------------------------------------
-- | Normalization

class Normal a where
  normal :: [String] -> a -> a

instance Normal Env where
  normal ns (Env (rho,vs,fs,os)) = Env (normal ns (rho,vs,fs,os))

instance Normal Val where
  normal ns v = case v of
    VU                  -> VU
    Ter (Lam x t u) e   ->
      let w = eval e t
          v@(VVar n _) = mkVarNice ns x w
      in VLam n (normal ns w) $ normal (n:ns) (eval (upd (x,v) e) u)
    Ter t e             -> Ter t (normal ns e)
    VPi u v             -> VPi (normal ns u) (normal ns v)
    VSigma u v          -> VSigma (normal ns u) (normal ns v)
    VPair u v           -> VPair (normal ns u) (normal ns v)
    VCon n us           -> VCon n (normal ns us)
    VPCon n u us phis   -> VPCon n (normal ns u) (normal ns us) phis
    VPathP a u0 u1      -> VPathP (normal ns a) (normal ns u0) (normal ns u1)
    VPLam i u           -> VPLam i (normal ns u)
    VComp u v vs        -> VComp (normal ns u) (normal ns v) (normal ns vs)
    VHComp u v vs       -> VHComp (normal ns u) (normal ns v) (normal ns vs)
    VGlue u equivs      -> VGlue (normal ns u) (normal ns equivs)
    VGlueElem u us      -> VGlueElem (normal ns u) (normal ns us)
    VUnGlueElem u us    -> VUnGlueElem (normal ns u) (normal ns us)
    VUnGlueElemU e u us -> VUnGlueElemU (normal ns e) (normal ns u) (normal ns us)
    VCompU a ts         -> VCompU (normal ns a) (normal ns ts)
    VVar x t            -> VVar x (normal ns t)
    VFst t              -> VFst (normal ns t)
    VSnd t              -> VSnd (normal ns t)
    VSplit u t          -> VSplit (normal ns u) (normal ns t)
    VApp u v            -> VApp (normal ns u) (normal ns v)
    VAppFormula u phi   -> VAppFormula (normal ns u) (normal ns phi)
    VId a u v           -> VId (normal ns a) (normal ns u) (normal ns v)
    VIdPair u us        -> VIdPair (normal ns u) (normal ns us)
    VIdJ a u c d x p    -> VIdJ (normal ns a) (normal ns u) (normal ns c)
                                (normal ns d) (normal ns x) (normal ns p)
    _                   -> v

instance Normal (C.Nameless a) where
  normal _ = id

instance Normal Ctxt where
  normal _ = id

instance Normal C.Formula where
  normal _ = C.fromDNF . C.dnf

instance Normal a => Normal (Map k a) where
  normal ns = Map.map (normal ns)

instance (Normal a,Normal b) => Normal (a,b) where
  normal ns (u,v) = (normal ns u,normal ns v)

```

```haskell
instance (Normal a,Normal b,Normal c) => Normal (a,b,c) where
  normal ns (u,v,w) = (normal ns u,normal ns v,normal ns w)

instance (Normal a,Normal b,Normal c,Normal d) => Normal (a,b,c,d) where
  normal ns (u,v,w,x) =
    (normal ns u,normal ns v,normal ns w, normal ns x)

instance Normal a => Normal [a] where
  normal ns = map (normal ns)
::::::::::::::
Main.hs
::::::::::::::
module Main where

import Control.Monad.Reader
import qualified Control.Exception as E
import Data.List
import Data.Time
import System.Directory
import System.FilePath
import System.Environment
import System.Console.GetOpt
import System.Console.Haskeline
import System.Console.Haskeline.History
import Text.Printf

import Exp.Lex
import Exp.Par
import Exp.Print
import Exp.Abs hiding (NoArg)
import Exp.Layout
import Exp.ErrM

-- | CubicalTT syntax
import CTT
-- | Resolver for symbol resolution.
import Resolver
-- | Type checker
import qualified TypeChecker as TC
-- | Evaluator
import qualified Eval as E

type Interpreter a = InputT IO a


-- | Flags
data Flag = Debug | Batch | Help | Version | Time
  deriving (Eq,Show)

options :: [OptDescr Flag]
options = [ Option "d"  ["debug"]   (NoArg Debug)   "run in debugging mode"
          , Option "b"  ["batch"]   (NoArg Batch)   "run in batch mode"
          , Option ""   ["help"]    (NoArg Help)    "print help"
          , Option "-t" ["time"]    (NoArg Time)    "measure time spent computing"
          , Option ""   ["version"] (NoArg Version) "print version number" ]


-- | Version number, welcome message, usage and prompt strings
version, welcome, usage, prompt :: String
version = "1.0"
welcome = "cubical, version: " ++ version ++ "  (:h for help)\n"
usage   = "Usage: cubical [options] <file.ctt>\nOptions:"
prompt  = "> "

-- | Entrypoint. handle command line arguments. If passed a file, load the file
-- and then enter REPL loop. If not, directly enter REPL loop.
main :: IO ()
main = do
  args <- getArgs
  case getOpt Permute options args of
```

```haskell
    (flags,files,[])
      | Help    `elem` flags -> putStrLn $ usageInfo usage options
      | Version `elem` flags -> putStrLn version
      | otherwise -> case files of
       []  -> do
         putStrLn welcome
         runInputT (settings []) (loop flags [] [] TC.verboseEnv)
       [f] -> do
         putStrLn welcome
         putStrLn $ "Loading " ++ show f
         initLoop flags f emptyHistory
       _   -> putStrLn $ "Input error: zero or one file expected\n\n" ++
                         usageInfo usage options
    (_,_,errs) -> putStrLn $ "Input error: " ++ concat errs ++ "\n" ++
                             usageInfo usage options

-- | The main loop
loop :: [Flag] -> FilePath -> [(CTT.Ident,SymKind)] -> TC.TEnv -> Interpreter ()
loop flags f names tenv = do
  input <- getInputLine prompt
  case input of
    Nothing    -> outputStrLn help >> loop flags f names tenv
    Just ":q"  -> return ()
    Just ":r"  -> getHistory >>= lift . initLoop flags f
    Just (':':'l':' ':str)
      | ' ' `elem` str -> do outputStrLn "Only one file allowed after :l"
                             loop flags f names tenv
      | otherwise      -> getHistory >>= lift . initLoop flags str
    Just (':':'c':'d':' ':str) -> do lift (setCurrentDirectory str)
                                     loop flags f names tenv
    Just ":h"  -> outputStrLn help >> loop flags f names tenv
    Just str'  ->
      let (msg,str,mod) = case str' of
            (':':'n':' ':str) ->
              ("NORMEVAL: ",str,E.normal [])
            str -> ("EVAL: ",str,id)
      in case pExp (lexer str) of
      Bad err -> outputStrLn ("Parse error: " ++ err) >> loop flags f names tenv
      -- | Resolve the expression
      Ok  exp ->
        case runResolver $ local (insertIdents names) $ resolveExp exp of
          Left  err  -> do outputStrLn ("Resolver failed: " ++ err)
                           loop flags f names tenv
          Right body -> do
            -- | KEY STEP: type check the expression
            x <- liftIO $ TC.runInfer tenv body
            case x of
              Left err -> do outputStrLn ("Could not type-check: " ++ err)
                             loop flags f names tenv
              Right _  -> do
                start <- liftIO getCurrentTime
                -- | KEY STEP: evaluate the expression.
                let e = mod $ E.eval (TC.env tenv) body
                -- |  Let's not crash if the evaluation raises an error:
                liftIO $ catch (putStrLn (msg ++ shrink (show e)))
                               -- (writeFile "examples/nunivalence3.ctt" (show e))
                               (\e -> putStrLn ("Exception: " ++
                                                show (e :: SomeException)))
                stop <- liftIO getCurrentTime
                -- | Compute time and print nicely if `-t` is used.
                let time = diffUTCTime stop start
                    secs = read (takeWhile (/='.') (init (show time)))
                    rest = read ('0':dropWhile (/='.') (init (show time)))
                    mins = secs `quot` 60
                    sec  = printf "%.3f" (fromInteger (secs `rem` 60) + rest :: Float)
                when (Time `elem` flags) $
                   outputStrLn $ "Time: " ++ show mins ++ "m" ++ sec ++ "s"
                -- Only print in seconds:
                -- when (Time `elem` flags) $ outputStrLn $ "Time: " ++ show time
                loop flags f names tenv
```

```haskell

-- | load file
initLoop :: [Flag] -> FilePath -> History -> IO ()
initLoop flags f hist = do
  -- Parse and type check files
  -- | imports defined below. Load modules. Module defined in ???
  (_,_,mods) <- E.catch (imports True ([],[],[]) f)
                        (\e -> do putStrLn $ unlines $
                                    ("Exception: " :
                                     (takeWhile (/= "CallStack (from HasCallStack):")
                                                   (lines $ show (e :: SomeException))))
                                  return ([],[],[]))
  -- | Translate to TT. resolveModules from from Resolver.hs.
  let res = runResolver $ resolveModules mods
  case res of
    Left err    -> do
      putStrLn $ "Resolver failed: " ++ err
      runInputT (settings []) (putHistory hist >> loop flags f [] TC.verboseEnv)
    Right (adefs,names) -> do
      -- After resolivng the file check if some definitions were shadowed:
      let ns = map fst names
          uns = nub ns
          dups = ns \\ uns
      unless (dups == []) $
        putStrLn $ "Warning: the following definitions were shadowed [" ++
                   intercalate ", " dups ++ "]"
      (merr,tenv) <- TC.runDeclss TC.verboseEnv adefs
      case merr of
        Just err -> putStrLn $ "Type checking failed: " ++ shrink err
        Nothing  -> unless (mods == []) $ putStrLn "File loaded."
      if Batch `elem` flags
        then return ()
        else -- Compute names for auto completion
             runInputT (settings [n | (n,_) <- names])
               (putHistory hist >> loop flags f names tenv)


-- | TODO: where is this coming from?!
lexer :: String -> [Token]
lexer = resolveLayout True . myLexer

showTree :: (Show a, Print a) => a -> IO ()
showTree tree = do
  putStrLn $ "\n[Abstract Syntax]\n\n" ++ show tree
  putStrLn $ "\n[Linearized tree]\n\n" ++ printTree tree

-- Used for auto completion
searchFunc :: [String] -> String -> [Completion]
searchFunc ns str = map simpleCompletion $ filter (str `isPrefixOf`) ns

settings :: [String] -> Settings IO
settings ns = Settings
  { historyFile    = Nothing
  , complete       = completeWord Nothing " \t" $ return . searchFunc ns
  , autoAddHistory = True }


shrink :: String -> String
shrink s = s -- if length s > 1000 then take 1000 s ++ "..." else s



-- | import modules
-- (not ok,loaded,already loaded defs) -> to load ->
--   (new not ok, new loaded, new defs)
-- the bool determines if it should be verbose or not
imports :: Bool -> ([String],[String],[Module]) -> String ->
           IO ([String],[String],[Module])
imports v st@(notok,loaded,mods) f
  | f `elem` notok  = error ("Looping imports in " ++ f)
```

```
  | f `elem` loaded = return st
  | otherwise       = do
    b <- doesFileExist f
    when (not b) $ error (f ++ " does not exist")
    let prefix = dropFileName f
    s <- readFile f
    let ts = lexer s
    -- | parse a module and store it.
    -- | check that module name is same as filename.
    case pModule ts of
      Bad s -> error ("Parse failed in " ++ show f ++ "\n" ++ show s)
      Ok mod@(Module (AIdent (_,name)) imp decls) -> do
        let imp_ctt = [prefix ++ i ++ ".ctt" | Import (AIdent (_,i)) <- imp]
        when (name /= dropExtension (takeFileName f)) $
          error ("Module name mismatch in " ++ show f ++ " with wrong name " ++ "\"" ++ name ++ "\"")
        (notok1,loaded1,mods1) <-
          foldM (imports v) (f:notok,loaded,mods) imp_ctt
        when v $ putStrLn $ "Parsed " ++ show f ++ " successfully!"
        return (notok,f:loaded1,mods1 ++ [mod])

help :: String
help = "\nAvailable commands:\n" ++
       "  <statement>     infer type and evaluate statement\n" ++
       "  :n <statement>  normalize statement\n" ++
       "  :q              quit\n" ++
       "  :l <filename>   loads filename (and resets environment before)\n" ++
       "  :cd <path>      change directory to path\n" ++
       "  :r              reload\n" ++
       "  :h              display this message\n"
::::::::::::::
Resolver.hs
::::::::::::::
{-# LANGUAGE TupleSections #-}

-- | Convert the concrete syntax into the syntax of cubical TT.
module Resolver(
     Resolver
   , resolveModule
   , resolveModules
   , SymKind(..)
   , runResolver
   , resolveExp
   , insertIdents) where

import Control.Applicative
import Control.Monad
import Control.Monad.Reader
import Control.Monad.Except
import Control.Monad.Identity
import Data.Maybe
import Data.List
import Data.Map (Map,(!))
import qualified Data.Map as Map

import Exp.Abs
import CTT (Ter,Ident,Loc(..),mkApps,mkWheres)
import qualified CTT
import Connections (negFormula,andFormula,orFormula)
import qualified Connections as C

-- | Useful auxiliary functions

-- Applicative cons
(<:>) :: Applicative f => f a -> f [a] -> f [a]
a <:> b = (:) <$> a <*> b

-- Un-something functions
unVar :: Exp -> Maybe Ident
unVar (Var (AIdent (_,x))) = Just x
```

```
unVar _                    = Nothing

unWhere :: ExpWhere -> Exp
unWhere (Where e ds) = Let ds e
unWhere (NoWhere e)  = e

-- Tail recursive form to transform a sequence of applications
-- App (App (App u v) ...) w  into (u, [v, …, w])
-- (cleaner than the previous version of unApps)
unApps :: Exp -> [Exp] -> (Exp, [Exp])
unApps (App u v) ws = unApps u (v : ws)
unApps u         ws = (u, ws)

-- Turns an expression of the form App (... (App id1 id2) ... idn)
-- into a list of idents
appsToIdents :: Exp -> Maybe [Ident]
appsToIdents = mapM unVar . uncurry (:) . flip unApps []

-- Transform a sequence of applications
-- (((u v1) .. vn) phi1) .. phim into (u,[v1,..,vn],[phi1,..,phim])
unAppsFormulas :: Exp -> [Formula]-> (Exp,[Exp],[Formula])
unAppsFormulas (AppFormula u phi) phis = unAppsFormulas u (phi:phis)
unAppsFormulas u phis = (x,xs,phis)
  where (x,xs) = unApps u []

-- Flatten a tele
flattenTele :: [Tele] -> [(Ident,Exp)]
flattenTele tele =
  [ (unAIdent i,typ) | Tele id ids typ <- tele, i <- id:ids ]

-- Flatten a PTele
flattenPTele :: [PTele] -> Resolver [(Ident,Exp)]
flattenPTele []                   = return []
flattenPTele (PTele exp typ : xs) = case appsToIdents exp of
  Just ids -> do
    pt <- flattenPTele xs
    return $ map (,typ) ids ++ pt
  Nothing -> throwError "malformed ptele"

-------------------------------------------------------------------------------
-- | Resolver and environment

data SymKind = Variable | Constructor | PConstructor | Name
  deriving (Eq,Show)

-- local environment for constructors
data Env = Env { envModule :: String,
                 variables :: [(Ident,SymKind)] }
  deriving (Eq,Show)

type Resolver a = ReaderT Env (Except String) a

emptyEnv :: Env
emptyEnv = Env "" []

runResolver :: Resolver a -> Either String a
runResolver x = runIdentity $ runExceptT $ runReaderT x emptyEnv

updateModule :: String -> Env -> Env
updateModule mod e = e{envModule = mod}

insertIdent :: (Ident,SymKind) -> Env -> Env
insertIdent (n,var) e
  | n == "_"  = e
  | otherwise = e{variables = (n,var) : variables e}

insertIdents :: [(Ident,SymKind)] -> Env -> Env
insertIdents = flip $ foldr insertIdent

insertName :: AIdent -> Env -> Env
```

```haskell
insertName (AIdent (_,x)) = insertIdent (x,Name)

insertNames :: [AIdent] -> Env -> Env
insertNames = flip $ foldr insertName

insertVar :: Ident -> Env -> Env
insertVar x = insertIdent (x,Variable)

insertVars :: [Ident] -> Env -> Env
insertVars = flip $ foldr insertVar

insertAIdent :: AIdent -> Env -> Env
insertAIdent (AIdent (_,x)) = insertIdent (x,Variable)

insertAIdents :: [AIdent] -> Env -> Env
insertAIdents  = flip $ foldr insertAIdent

getLoc :: (Int,Int) -> Resolver Loc
getLoc l = Loc <$> asks envModule <*> pure l

unAIdent :: AIdent -> Ident
unAIdent (AIdent (_,x)) = x

resolveName :: AIdent -> Resolver C.Name
resolveName (AIdent (l,x)) = do
  modName <- asks envModule
  vars <- asks variables
  case lookup x vars of
    Just Name -> return $ C.Name x
    _ -> throwError $ "Cannot resolve name " ++ x ++ " at position " ++
                      show l ++ " in module " ++ modName

resolveVar :: AIdent -> Resolver Ter
resolveVar (AIdent (l,x)) = do
  modName <- asks envModule
  vars    <- asks variables
  case lookup x vars of
    Just Variable    -> return $ CTT.Var x
    Just Constructor -> return $ CTT.Con x []
    Just PConstructor ->
      throwError $ "The path constructor " ++ x ++ " is used as a" ++
                   " variable at " ++ show l ++ " in " ++ modName ++
                   " (path constructors should have their type in" ++
                   " curly braces as first argument)"
    Just Name        ->
      throwError $ "Name " ++ x ++ " used as a variable at position " ++
                   show l ++ " in module " ++ modName
    _ -> throwError $ "Cannot resolve variable " ++ x ++ " at position " ++
                      show l ++ " in module " ++ modName

lam :: (Ident,Exp) -> Resolver Ter -> Resolver Ter
lam (a,t) e = CTT.Lam a <$> resolveExp t <*> local (insertVar a) e

lams :: [(Ident,Exp)] -> Resolver Ter -> Resolver Ter
lams = flip $ foldr lam

plam :: AIdent -> Resolver Ter -> Resolver Ter
plam i e = CTT.PLam (C.Name (unAIdent i)) <$> local (insertName i) e

plams :: [AIdent] -> Resolver Ter -> Resolver Ter
plams [] _ = throwError "Empty plam abstraction"
plams xs e = foldr plam e xs

bind :: (Ter -> Ter) -> (Ident,Exp) -> Resolver Ter -> Resolver Ter
bind f (x,t) e = f <$> lam (x,t) e

binds :: (Ter -> Ter) -> [(Ident,Exp)] -> Resolver Ter -> Resolver Ter
binds f = flip $ foldr $ bind f

resolveApps :: Exp -> [Exp] -> Resolver Ter
```

```haskell
resolveApps x xs = mkApps <$> resolveExp x <*> mapM resolveExp xs

resolveExp :: Exp -> Resolver Ter
resolveExp e = case e of
  U             -> return CTT.U
  Var x         -> resolveVar x
  App t s       -> resolveApps x xs
    where (x,xs) = unApps t [s]
  Sigma ptele b -> do
    tele <- flattenPTele ptele
    binds CTT.Sigma tele (resolveExp b)
  Pi ptele b    -> do
    tele <- flattenPTele ptele
    binds CTT.Pi tele (resolveExp b)
  Fun a b       -> bind CTT.Pi ("_",a) (resolveExp b)
  Lam ptele t   -> do
    tele <- flattenPTele ptele
    lams tele (resolveExp t)
  Fst t         -> CTT.Fst <$> resolveExp t
  Snd t         -> CTT.Snd <$> resolveExp t
  Pair t0 ts    -> do
    e  <- resolveExp t0
    es <- mapM resolveExp ts
    return $ foldr1 CTT.Pair (e:es)
  Split t brs -> do
    t'   <- resolveExp t
    brs' <- mapM resolveBranch brs
    l@(Loc n (i,j)) <- getLoc (case brs of
                                  OBranch (AIdent (l,_)) _ _:_ -> l
                                  PBranch (AIdent (l,_)) _ _ _:_ -> l
                                  _ -> (0,0))
    return $ CTT.Split (n ++ "_L" ++ show i ++ "_C" ++ show j) l t' brs'
  Let decls e   -> do
    (rdecls,names) <- resolveDecls decls
    mkWheres rdecls <$> local (insertIdents names) (resolveExp e)
  PLam is e     -> plams is (resolveExp e)
  Hole (HoleIdent (l,_)) -> CTT.Hole <$> getLoc l
  AppFormula t phi ->
    let (x,xs,phis) = unAppsFormulas e []
    in case x of
      PCon n a ->
        CTT.PCon (unAIdent n) <$> resolveExp a <*> mapM resolveExp xs
                              <*> mapM resolveFormula phis
      _ -> CTT.AppFormula <$> resolveExp t <*> resolveFormula phi
  PathP a u v   -> CTT.PathP <$> resolveExp a <*> resolveExp u <*> resolveExp v
  Comp u v ts   -> CTT.Comp <$> resolveExp u <*> resolveExp v <*> resolveSystem ts
  HComp u v ts  -> CTT.HComp <$> resolveExp u <*> resolveExp v <*> resolveSystem ts
  Fill u v ts   -> CTT.Fill <$> resolveExp u <*> resolveExp v <*> resolveSystem ts
  Trans u v     -> CTT.Comp <$> resolveExp u <*> resolveExp v <*> pure Map.empty
  Glue u ts     -> CTT.Glue <$> resolveExp u <*> resolveSystem ts
  GlueElem u ts -> CTT.GlueElem <$> resolveExp u <*> resolveSystem ts
  UnGlueElem u ts -> CTT.UnGlueElem <$> resolveExp u <*> resolveSystem ts
  Id a u v      -> CTT.Id <$> resolveExp a <*> resolveExp u <*> resolveExp v
  IdPair u ts   -> CTT.IdPair <$> resolveExp u <*> resolveSystem ts
  IdJ a t c d x p -> CTT.IdJ <$> resolveExp a <*> resolveExp t <*> resolveExp c
                             <*> resolveExp d <*> resolveExp x <*> resolveExp p
  _ -> do
    modName <- asks envModule
    throwError ("Could not resolve " ++ show e ++ " in module " ++ modName)

resolveWhere :: ExpWhere -> Resolver Ter
resolveWhere = resolveExp . unWhere

resolveSystem :: System -> Resolver (C.System Ter)
resolveSystem (System ts) = do
  ts' <- sequence [ (,) <$> resolveFace alpha <*> resolveExp u
                  | Side alpha u <- ts ]
  let alphas = map fst ts'
  unless (nub alphas == alphas) $
    throwError $ "system contains same face multiple times: " ++
```

```
                C.showListSystem ts'
  -- Note: the symbols in alpha are in scope in u, but they mean 0 or 1
  return $ Map.fromList ts'

resolveFace :: [Face] -> Resolver C.Face
resolveFace alpha =
  Map.fromList <$> sequence [ (,) <$> resolveName i <*> resolveDir d
                            | Face i d <- alpha ]

resolveDir :: Dir -> Resolver C.Dir
resolveDir Dir0 = return 0
resolveDir Dir1 = return 1

resolveFormula :: Formula -> Resolver C.Formula
resolveFormula (Dir d)          = C.Dir <$> resolveDir d
resolveFormula (Atom i)         = C.Atom <$> resolveName i
resolveFormula (Neg phi)        = negFormula <$> resolveFormula phi
resolveFormula (Conj phi _ psi) =
    andFormula <$> resolveFormula phi <*> resolveFormula psi
resolveFormula (Disj phi psi)   =
    orFormula <$> resolveFormula phi <*> resolveFormula psi

resolveBranch :: Branch -> Resolver CTT.Branch
resolveBranch (OBranch (AIdent (_,lbl)) args e) = do
  re <- local (insertAIdents args) $ resolveWhere e
  return $ CTT.OBranch lbl (map unAIdent args) re
resolveBranch (PBranch (AIdent (_,lbl)) args is e) = do
  re <- local (insertNames is . insertAIdents args) $ resolveWhere e
  let names = map (C.Name . unAIdent) is
  return $ CTT.PBranch lbl (map unAIdent args) names re

resolveTele :: [(Ident,Exp)] -> Resolver CTT.Tele
resolveTele []        = return []
resolveTele ((i,d):t) =
  ((i,) <$> resolveExp d) <:> local (insertVar i) (resolveTele t)

resolveLabel :: [(Ident,SymKind)] -> Label -> Resolver CTT.Label
resolveLabel _ (OLabel n vdecl) =
  CTT.OLabel (unAIdent n) <$> resolveTele (flattenTele vdecl)
resolveLabel cs (PLabel n vdecl is sys) = do
  let tele' = flattenTele vdecl
      ts    = map fst tele'
      names = map (C.Name . unAIdent) is
      n'    = unAIdent n
      cs'   = delete (n',PConstructor) cs
  CTT.PLabel n' <$> resolveTele tele' <*> pure names
               <*> local (insertNames is . insertIdents cs' . insertVars ts)
                      (resolveSystem sys)

-- Resolve a non-mutual declaration; returns resolver for type and
-- body separately
resolveNonMutualDecl :: Decl -> (Ident,Resolver CTT.Ter
                                ,Resolver CTT.Ter,[(Ident,SymKind)])
resolveNonMutualDecl d = case d of
  DeclDef (AIdent (_,f)) tele t body ->
    let tele' = flattenTele tele
        a     = binds CTT.Pi tele' (resolveExp t)
        d     = lams tele' (local (insertVar f) $ resolveWhere body)
    in (f,a,d,[(f,Variable)])
  DeclData x tele sums  -> resolveDeclData x tele sums null
  DeclHData x tele sums ->
    resolveDeclData x tele sums (const False) -- always pick HSum
  DeclSplit (AIdent (l,f)) tele t brs ->
    let tele' = flattenTele tele
        vars  = map fst tele'
        a     = binds CTT.Pi tele' (resolveExp t)
        d     = do
                  loc <- getLoc l
                  ty  <- local (insertVars vars) $ resolveExp t
                  brs' <- local (insertVars (f:vars)) (mapM resolveBranch brs)
```

```haskell
                  lams tele' (return $ CTT.Split f loc ty brs')
    in (f,a,d,[(f,Variable)])
  DeclUndef (AIdent (l,f)) tele t ->
    let tele' = flattenTele tele
        a     = binds CTT.Pi tele' (resolveExp t)
        d     = CTT.Undef <$> getLoc l <*> a
    in (f,a,d,[(f,Variable)])

-- Helper function to resolve data declarations. The predicate p is
-- used to decide if we should use Sum or HSum.
resolveDeclData :: AIdent -> [Tele] -> [Label] -> ([(Ident,SymKind)] -> Bool) ->
                   (Ident, Resolver Ter, Resolver Ter, [(Ident, SymKind)])
resolveDeclData (AIdent (l,f)) tele sums p =
  let tele' = flattenTele tele
      a     = binds CTT.Pi tele' (return CTT.U)
      cs    = [ (unAIdent lbl,Constructor) | OLabel lbl _ <- sums ]
      pcs   = [ (unAIdent lbl,PConstructor) | PLabel lbl _ _ _ <- sums ]
      sum   = if p pcs then CTT.Sum else CTT.HSum
      d = lams tele' $ local (insertVar f) $
            sum <$> getLoc l <*> pure f
                <*> mapM (resolveLabel (cs ++ pcs)) sums
  in (f,a,d,(f,Variable):cs ++ pcs)

resolveRTele :: [Ident] -> [Resolver CTT.Ter] -> Resolver CTT.Tele
resolveRTele [] _ = return []
resolveRTele (i:is) (t:ts) = do
  a  <- t
  as <- local (insertVar i) (resolveRTele is ts)
  return ((i,a):as)

-- Best effort to find the location of a declaration. This implementation
-- returns the location of the first identifier it contains.
findDeclLoc :: Decl -> Resolver Loc
findDeclLoc d = getLoc loc
    where loc = fromMaybe (-1, 0) $ mloc d
          mloc d = case d of
            DeclDef (AIdent (l, _)) _ _ _   -> Just l
            DeclData (AIdent (l, _)) _ _    -> Just l
            DeclHData (AIdent (l, _)) _ _   -> Just l
            DeclSplit (AIdent (l, _)) _ _ _ -> Just l
            DeclUndef (AIdent (l, _)) _ _   -> Just l
            DeclMutual ds                   -> listToMaybe $ mapMaybe mloc ds
            DeclOpaque (AIdent (l, _))      -> Just l
            DeclTransparent (AIdent (l, _)) -> Just l
            DeclTransparentAll              -> Nothing

-- Resolve a declaration
resolveDecl :: Decl -> Resolver (CTT.Decls,[(Ident,SymKind)])
resolveDecl d = case d of
  DeclMutual decls -> do
    let (fs,ts,bs,nss) = unzip4 $ map resolveNonMutualDecl decls
        ns = concat nss -- TODO: some sanity checks? Duplicates!?
    when (nub (map fst ns) /= concatMap (map fst) nss) $
      throwError ("Duplicated constructor or ident: " ++ show nss)
    as <- resolveRTele fs ts
    -- The bodies know about all the names and constructors in the
    -- mutual block
    ds <- sequence $ map (local (insertIdents ns)) bs
    let ads = zipWith (\ (x,y) z -> (x,(y,z))) as ds
    l <- findDeclLoc d
    return (CTT.MutualDecls l ads,ns)
  DeclOpaque i  -> do
    resolveVar i
    return (CTT.OpaqueDecl (unAIdent i), [])
  DeclTransparent i -> do
    resolveVar i
    return (CTT.TransparentDecl (unAIdent i), [])
  DeclTransparentAll -> return (CTT.TransparentAllDecl, [])
  _ -> do let (f,typ,body,ns) = resolveNonMutualDecl d
          l <- findDeclLoc d
```

```
          a <- typ
          d <- body
          return (CTT.MutualDecls l [(f,(a,d))],ns)

resolveDecls :: [Decl] -> Resolver ([CTT.Decls],[(Ident,SymKind)])
resolveDecls []     = return ([],[])
resolveDecls (d:ds) = do
  (rtd,names)  <- resolveDecl d
  (rds,names') <- local (insertIdents names) $ resolveDecls ds
  return (rtd : rds, names' ++ names)

resolveModule :: Module -> Resolver ([CTT.Decls],[(Ident,SymKind)])
resolveModule (Module (AIdent (_,n)) _ decls) =
  local (updateModule n) $ resolveDecls decls

resolveModules :: [Module] -> Resolver ([CTT.Decls],[(Ident,SymKind)])
resolveModules []         = return ([],[])
resolveModules (mod:mods) = do
  (rmod, names)  <- resolveModule mod
  (rmods,names') <- local (insertIdents names) $ resolveModules mods
  return (rmod ++ rmods, names' ++ names)
::::::::::::::
Setup.hs
::::::::::::::
import Distribution.Simple
import Distribution.Simple.Program
import System.Process (system)

main :: IO ()
main = defaultMainWithHooks $ simpleUserHooks {
  hookedPrograms = [bnfc],
  preBuild = \args buildFlags -> do
      _ <- system "bnfc --haskell -d Exp.cf"
      preBuild simpleUserHooks args buildFlags
}

bnfc :: Program
bnfc = (simpleProgram "bnfc") {
    programFindVersion = findProgramVersion "--version" id
  }
::::::::::::::
TypeChecker.hs
::::::::::::::
{-# LANGUAGE TupleSections #-}
module TypeChecker(
  runInfer
  , verboseEnv
  , runDeclss
  , env
  , TEnv) where

import Control.Applicative hiding (empty)
import Control.Monad
import Control.Monad.Except
import Control.Monad.Reader
import Data.Map (Map,(!),mapWithKey,assocs,filterWithKey,elems,keys
                ,intersection,intersectionWith,intersectionWithKey
                ,toList,fromList)
import qualified Data.Map as Map
import qualified Data.Traversable as T

import CTT
import qualified Connections as C
import qualified Eval as E

-- | Type checking monad
type Typing a = ReaderT TEnv (ExceptT String IO) a

-- | Environment for type checker
data TEnv =
```

```haskell
  TEnv { names   :: [String] -- generated names
       , indent  :: Int
       , env     :: Env
       , verbose :: Bool  -- Should it be verbose and print what it typechecks?
       } deriving (Eq)

verboseEnv, silentEnv :: TEnv
verboseEnv = TEnv [] 0 emptyEnv True
silentEnv  = TEnv [] 0 emptyEnv False

-- Trace function that depends on the verbosity flag
trace :: String -> Typing ()
trace s = do
  b <- asks verbose
  when b $ liftIO (putStrLn s)

-------------------------------------------------------------------------------
-- | Functions for running computations in the type checker monad

runTyping :: TEnv -> Typing a -> IO (Either String a)
runTyping env t = runExceptT $ runReaderT t env

runDecls :: TEnv -> Decls -> IO (Either String TEnv)
runDecls tenv d = runTyping tenv $ do
  checkDecls d
  return $ addDecls d tenv

runDeclss :: TEnv -> [Decls] -> IO (Maybe String,TEnv)
runDeclss tenv []     = return (Nothing, tenv)
runDeclss tenv (d:ds) = do
  x <- runDecls tenv d
  case x of
    Right tenv' -> runDeclss tenv' ds
    Left s      -> return (Just s, tenv)

runInfer :: TEnv -> Ter -> IO (Either String Val)
runInfer lenv e = runTyping lenv (infer e)

-------------------------------------------------------------------------------
-- | Modifiers for the environment

addTypeVal :: (Ident,Val) -> TEnv -> TEnv
addTypeVal (x,a) (TEnv ns ind rho v) =
  let w@(VVar n _) = mkVarNice ns x a
  in TEnv (n:ns) ind (upd (x,w) rho) v

addSub :: (C.Name,C.Formula) -> TEnv -> TEnv
addSub iphi (TEnv ns ind rho v) = TEnv ns ind (sub iphi rho) v

addSubs :: [(C.Name,C.Formula)] -> TEnv -> TEnv
addSubs = flip $ foldr addSub

addType :: (Ident,Ter) -> TEnv -> TEnv
addType (x,a) tenv@(TEnv _ _ rho _) = addTypeVal (x,E.eval rho a) tenv

addBranch :: [(Ident,Val)] -> Env -> TEnv -> TEnv
addBranch nvs env (TEnv ns ind rho v) =
  TEnv ([n | (_,VVar n _) <- nvs] ++ ns) ind (upds nvs rho) v

addDecls :: Decls -> TEnv -> TEnv
addDecls d (TEnv ns ind rho v) = TEnv ns ind (def d rho) v

addTele :: Tele -> TEnv -> TEnv
addTele xas lenv = foldl (flip addType) lenv xas

faceEnv :: C.Face -> TEnv -> TEnv
faceEnv alpha tenv = tenv{env=env tenv `C.face` alpha}

-------------------------------------------------------------------------------
-- | Various useful functions
```

```haskell

-- Extract the type of a label as a closure
getLblType :: LIdent -> Val -> Typing (Tele, Env)
getLblType c (Ter (Sum _ _ cas) r) = case lookupLabel c cas of
  Just as -> return (as,r)
  Nothing -> throwError ("getLblType: " ++ show c ++ " in " ++ show cas)
getLblType c (Ter (HSum _ _ cas) r) = case lookupLabel c cas of
  Just as -> return (as,r)
  Nothing -> throwError ("getLblType: " ++ show c ++ " in " ++ show cas)
getLblType c u = throwError ("expected a data type for the constructor "
                             ++ c ++ " but got " ++ show u)

-- Monadic version of unless
unlessM :: Monad m => m Bool -> m () -> m ()
unlessM mb x = mb >>= flip unless x

mkVars :: [String] -> Tele -> Env -> [(Ident,Val)]
mkVars _ [] _           = []
mkVars ns ((x,a):xas) nu =
  let w@(VVar n _) = mkVarNice ns x (E.eval nu a)
  in (x,w) : mkVars (n:ns) xas (upd (x,w) nu)

-- Test if two values are convertible
(===) :: E.Convertible a => a -> a -> Typing Bool
u === v = E.conv <$> asks names <*> pure u <*> pure v

-- eval in the typing monad
evalTyping :: Ter -> Typing Val
evalTyping t = E.eval <$> asks env <*> pure t

-------------------------------------------------------------------------------
-- | The bidirectional type checker

-- Check that t has type a
check :: Val -> Ter -> Typing ()
check a t = case (a,t) of
  (_,Undef{}) -> return ()
  (_,Hole l)  -> do
      rho <- asks env
      let e = unlines (reverse (contextOfEnv rho))
      ns <- asks names
      trace $ "\nHole at " ++ show l ++ ":\n\n" ++
              e ++ replicate 80 '-' ++ "\n" ++ show (E.normal ns a)  ++ "\n"
  (_,Con c es) -> do
    (bs,nu) <- getLblType c a
    checks (bs,nu) es
  (VU,Pi f)       -> checkFam f
  (VU,Sigma f)    -> checkFam f
  (VU,Sum _ _ bs) -> forM_ bs $ \lbl -> case lbl of
    OLabel _ tele -> checkTele tele
    PLabel _ tele is ts ->
      throwError $ "check: no path constructor allowed in " ++ show t
  (VU,HSum _ _ bs) -> forM_ bs $ \lbl -> case lbl of
    OLabel _ tele -> checkTele tele
    PLabel _ tele is ts -> do
      checkTele tele
      rho <- asks env
      unless (all (`elem` is) (C.domain ts)) $
        throwError "names in path label C.System" -- TODO
      mapM_ checkFresh is
      let iis = zip is (map C.Atom is)
      local (addSubs iis . addTele tele) $ do
        checkSystemWith ts $ \alpha talpha ->
          local (faceEnv alpha) $
            -- NB: the type doesn't depend on is
            check (Ter t rho) talpha
        rho' <- asks env
        checkCompSystem (E.evalSystem rho' ts)
  (VPi va@(Ter (Sum _ _ cas) nu) f,Split _ _ ty ces) -> do
    check VU ty
```

```haskell
      rho <- asks env
      unlessM (a === E.eval rho ty) $ throwError "check: split annotations"
      if map labelName cas == map branchName ces
         then sequence_ [ checkBranch (lbl,nu) f brc (Ter t rho) va
                        | (brc, lbl) <- zip ces cas ]
         else throwError "case branches does not match the data type"
    (VPi va@(Ter (HSum _ _ cas) nu) f,Split _ _ ty ces) -> do
      check VU ty
      rho <- asks env
      unlessM (a === E.eval rho ty) $ throwError "check: split annotations"
      if map labelName cas == map branchName ces
         then sequence_ [ checkBranch (lbl,nu) f brc (Ter t rho) va
                        | (brc, lbl) <- zip ces cas ]
         else throwError "case branches does not match the data type"
    (VPi a f,Lam x a' t)  -> do
      check VU a'
      ns <- asks names
      rho <- asks env
      unlessM (a === E.eval rho a') $
        throwError $ "check: lam types don't match"
          ++ "\nlambda type annotation: " ++ show a'
          ++ "\ndomain of Pi: " ++ show a
          ++ "\nnormal form of type: " ++ show (E.normal ns a)
      let var = mkVarNice ns x a

      local (addTypeVal (x,a)) $ check (E.app f var) t
    (VSigma a f, Pair t1 t2) -> do
      check a t1
      v <- evalTyping t1
      check (E.app f v) t2
    (_,Where e d) -> do
      local (\tenv@TEnv{indent=i} -> tenv{indent=i + 2}) $ checkDecls d
      local (addDecls d) $ check a e
    (VU,PathP a e0 e1) -> do
      (a0,a1) <- checkPLam (constPath VU) a
      check a0 e0
      check a1 e1
    (VPathP p a0 a1,PLam _ e) -> do
      (u0,u1) <- checkPLam p t
      ns <- asks names
      unless (E.conv ns a0 u0 && E.conv ns a1 u1) $
        throwError $ "path endpoints don't match for " ++ show e ++ ", got " ++
                     show (u0,u1) ++ ", but expected " ++ show (a0,a1)
    (VU,Glue a ts) -> do
      check VU a
      rho <- asks env
      checkGlue (E.eval rho a) ts
    (VGlue va ts,GlueElem u us) -> do
      check va u
      vu <- evalTyping u
      checkGlueElem vu ts us
    (VCompU va ves,GlueElem u us) -> do
      check va u
      vu <- evalTyping u
      checkGlueElemU vu ves us
    (VU,Id a a0 a1) -> do
      check VU a
      va <- evalTyping a
      check va a0
      check va a1
    (VId va va0 va1,IdPair w ts) -> do
      check (VPathP (constPath va) va0 va1) w
      vw <- evalTyping w
      checkSystemWith ts $ \alpha tAlpha ->
        local (faceEnv alpha) $ do
          check (va `C.face` alpha) tAlpha
          vtAlpha <- evalTyping tAlpha
          unlessM (vw `C.face` alpha === constPath vtAlpha) $
            throwError "malformed eqC"
      rho <- asks env
```

```haskell
    checkCompSystem (E.evalSystem rho ts) -- Not needed
  _ -> do
    v <- infer t
    unlessM (v === a) $
      throwError $ "check conv:\n" ++ show v ++ "\n/=\n" ++ show a

-- Check a list of declarations
checkDecls :: Decls -> Typing ()
checkDecls (MutualDecls _ []) = return ()
checkDecls (MutualDecls l d)  = do
  a <- asks env
  let (idents,tele,ters) = (declIdents d,declTele d,declTers d)
  ind <- asks indent
  trace (replicate ind ' ' ++ "Checking: " ++ unwords idents)
  checkTele tele
  local (addDecls (MutualDecls l d)) $ do
    rho <- asks env
    checks (tele,rho) ters
checkDecls (OpaqueDecl _)      = return ()
checkDecls (TransparentDecl _) = return ()
checkDecls TransparentAllDecl  = return ()

-- Check a telescope
checkTele :: Tele -> Typing ()
checkTele []          = return ()
checkTele ((x,a):xas) = do
  check VU a
  local (addType (x,a)) $ checkTele xas

-- Check a family
checkFam :: Ter -> Typing ()
checkFam (Lam x a b) = do
  check VU a
  local (addType (x,a)) $ check VU b
checkFam x = throwError $ "checkFam: " ++ show x

-- Check that a C.System is compatible
checkCompSystem :: C.System Val -> Typing ()
checkCompSystem vus = do
  ns <- asks names
  unless (E.isCompSystem ns vus)
    (throwError $ "Incompatible System " ++ C.showSystem vus)

-- Check the values at corresponding faces with a function, assumes
-- systems have the same faces
checkSystemsWith :: C.System a -> C.System b -> (C.Face -> a -> b -> Typing c) ->
                    Typing ()
checkSystemsWith us vs f = sequence_ $ elems $ intersectionWithKey f us vs

-- Check the faces of a C.System
checkSystemWith :: C.System a -> (C.Face -> a -> Typing b) -> Typing ()
checkSystemWith us f = sequence_ $ elems $ mapWithKey f us

-- Check a glueElem
checkGlueElem :: Val -> C.System Val -> C.System Ter -> Typing ()
checkGlueElem vu ts us = do
  unless (keys ts == keys us)
    (throwError ("Keys don't match in " ++ show ts ++ " and " ++ show us))
  rho <- asks env
  checkSystemsWith ts us
    (\alpha vt u -> local (faceEnv alpha) $ check (E.equivDom vt) u)
  let vus = E.evalSystem rho us
  checkSystemsWith ts vus (\alpha vt vAlpha ->
    unlessM (E.app (E.equivFun vt) vAlpha === (vu `C.face` alpha)) $
      throwError $ "Image of glue component " ++ show vAlpha ++
                   " doesn't match " ++ show vu)
  checkCompSystem vus

-- Check a glueElem against VComp _ ves
checkGlueElemU :: Val -> C.System Val -> C.System Ter -> Typing ()
```

```
checkGlueElemU vu ves us = do
  unless (keys ves == keys us)
    (throwError ("Keys don't match in " ++ show ves ++ " and " ++ show us))
  rho <- asks env
  checkSystemsWith ves us
    (\alpha ve u -> local (faceEnv alpha) $ check (ve E.@@ C.One) u)
  let vus = E.evalSystem rho us
  checkSystemsWith ves vus (\alpha ve vAlpha ->
    unlessM (E.eqFun ve vAlpha === (vu `C.face` alpha)) $
      throwError $ "Transport of glueElem (for compU) component " ++ show vAlpha ++
                   " doesn't match " ++ show vu)
  checkCompSystem vus

checkGlue :: Val -> C.System Ter -> Typing ()
checkGlue va ts = do
  checkSystemWith ts (\alpha tAlpha -> checkEquiv (va `C.face` alpha) tAlpha)
  rho <- asks env
  checkCompSystem (E.evalSystem rho ts)

-- An iso for a type b is a five-tuple: (a,f,g,s,t)   where
--  a : U
--  f : a -> b
--  g : b -> a
--  s : forall (y : b), f (g y) = y
--  t : forall (x : a), g (f x) = x
mkIso :: Val -> Val
mkIso vb = E.eval rho $
  Sigma $ Lam "a" U $
  Sigma $ Lam "f" (Pi (Lam "_" a b)) $
  Sigma $ Lam "g" (Pi (Lam "_" b a)) $
  Sigma $ Lam "s" (Pi (Lam "y" b $ PathP (PLam (C.Name "_") b) (App f (App g y)) y)) $
    Pi (Lam "x" a $ PathP (PLam (C.Name "_") a) (App g (App f x)) x)
  where [a,b,f,g,x,y] = map Var ["a","b","f","g","x","y"]
        rho = upd ("b",vb) emptyEnv

-- An equivalence for a type a is a triple (t,f,p) where
-- t : U
-- f : t -> a
-- p : (x : a) -> isContr ((y:t) * Id a x (f y))
-- with isContr c = (z : c) * ((z' : C) -> Id c z z')
mkEquiv :: Val -> Val
mkEquiv va = E.eval rho $
  Sigma $ Lam "t" U $
  Sigma $ Lam "f" (Pi (Lam "_" t a)) $
  Pi (Lam "x" a $ iscontrfib)
  where [a,b,f,x,y,s,t,z] = map Var ["a","b","f","x","y","s","t","z"]
        rho = upd ("a",va) emptyEnv
        fib = Sigma $ Lam "y" t (PathP (PLam (C.Name "_") a) x (App f y))
        iscontrfib = Sigma $ Lam "s" fib $
                     Pi $ Lam "z" fib $ PathP (PLam (C.Name "_") fib) s z

checkEquiv :: Val -> Ter -> Typing ()
checkEquiv va equiv = check (mkEquiv va) equiv

checkIso :: Val -> Ter -> Typing ()
checkIso vb iso = check (mkIso vb) iso

checkBranch :: (Label,Env) -> Val -> Branch -> Val -> Val -> Typing ()
checkBranch (OLabel _ tele,nu) f (OBranch c ns e) _ _ = do
  ns' <- asks names
  let us = map snd $ mkVars ns' tele nu
  local (addBranch (zip ns us) nu) $ check (E.app f (VCon c us)) e
checkBranch (PLabel _ tele is ts,nu) f (PBranch c ns js e) g va = do
  ns' <- asks names
  -- mapM_ checkFresh js
  let us   = mkVars ns' tele nu
      vus  = map snd us
      js'  = map C.Atom js
      vts  = E.evalSystem (subs (zip is js') (upds us nu)) ts
      vgts = intersectionWith E.app (C.border g vts) vts
```

```haskell
  local (addSubs (zip js js') . addBranch (zip ns vus) nu) $ do
    check (E.app f (VPCon c va vus js')) e
    ve  <- evalTyping e -- TODO: combine with next two lines?
    let veborder = C.border ve vts
    unlessM (veborder === vgts) $
      throwError $ "Faces in branch for " ++ show c ++ " don't match:"
                   ++ "\ngot\n" ++ C.showSystem veborder ++ "\nbut expected\n"
                   ++ C.showSystem vgts

checkFormula :: C.Formula -> Typing ()
checkFormula phi = do
  rho <- asks env
  let dom = domainEnv rho
  unless (all (`elem` dom) (C.support phi)) $
    throwError $ "checkFormula: " ++ show phi

checkFresh :: C.Name -> Typing ()
checkFresh i = do
  rho <- asks env
  when (i `elem` C.support rho)
    (throwError $ show i ++ " is already declared")

-- Check that a term is a PLam and output the source and target
checkPLam :: Val -> Ter -> Typing (Val,Val)
checkPLam v (PLam i a) = do
  rho <- asks env
  -- checkFresh i
  local (addSub (i,C.Atom i)) $ check (v E.@@ i) a
  return (E.eval (sub (i,C.Dir 0) rho) a, E.eval (sub (i,C.Dir 1) rho) a)
checkPLam v t = do
  vt <- infer t
  case vt of
    VPathP a a0 a1 -> do
      unlessM (a === v) $ throwError (
        "checkPLam\n" ++ show v ++ "\n/=\n" ++ show a)
      return (a0,a1)
    _ -> throwError $ show vt ++ " is not a path"

-- Return C.System such that:
--   rhoalpha |- p_alpha : Id (va alpha) (t0 rhoalpha) ualpha
-- Moreover, check that the C.System ps is compatible.
checkPLamSystem :: Ter -> Val -> C.System Ter -> Typing (C.System Val)
checkPLamSystem t0 va ps = do
  rho <- asks env
  v <- T.sequence $ mapWithKey (\alpha pAlpha ->
    local (faceEnv alpha) $ do
      rhoAlpha <- asks env
      (a0,a1)  <- checkPLam (va `C.face` alpha) pAlpha
      unlessM (a0 === E.eval rhoAlpha t0) $
        throwError $ "Incompatible C.System " ++ C.showSystem ps ++
                     ", component\n " ++ show pAlpha ++
                     "\nincompatible with\n " ++ show t0 ++
                     "\na0 = " ++ show a0 ++
                     "\nt0alpha = " ++ show (E.eval rhoAlpha t0) ++
                     "\nva = " ++ show va
      return a1) ps
  checkCompSystem (E.evalSystem rho ps)
  return v

checks :: (Tele,Env) -> [Ter] -> Typing ()
checks ([],_)         []     = return ()
checks ((x,a):xas,nu) (e:es) = do
  check (E.eval nu a) e
  v' <- evalTyping e
  checks (xas,upd (x,v') nu) es
checks _              _      =
  throwError "checks: incorrect number of arguments"

-- | infer the type of e
infer :: Ter -> Typing Val
```

```haskell
infer e = case e of
  U           -> return VU  -- U : U
  Var n       -> E.lookType n <$> asks env
  App t u -> do
    c <- infer t
    case c of
      VPi a f -> do
        check a u
        v <- evalTyping u
        return $ E.app f v
      _       -> throwError $ show c ++ " is not a product"
  Fst t -> do
    c <- infer t
    case c of
      VSigma a f -> return a
      _          -> throwError $ show c ++ " is not a sigma-type"
  Snd t -> do
    c <- infer t
    case c of
      VSigma a f -> do
        v <- evalTyping t
        return $ E.app f (E.fstVal v)
      _          -> throwError $ show c ++ " is not a sigma-type"
  Where t d -> do
    checkDecls d
    local (addDecls d) $ infer t
  UnGlueElem e _ -> do
    t <- infer e
    case t of
     VGlue a _ -> return a
     _ -> throwError (show t ++ " is not a Glue")
  AppFormula e phi -> do
    checkFormula phi
    t <- infer e
    case t of
      VPathP a _ _ -> return $ a E.@@ phi
      _ -> throwError (show e ++ " is not a path")
  Comp a t0 ps -> do
    (va0, va1) <- checkPLam (constPath VU) a
    va <- evalTyping a
    check va0 t0
    checkPLamSystem t0 va ps
    return va1
  HComp a u0 us -> do
    check VU a
    va <- evalTyping a
    check va u0
    checkPLamSystem u0 (constPath va) us
    return va
  Fill a t0 ps -> do
    (va0, va1) <- checkPLam (constPath VU) a
    va <- evalTyping a
    check va0 t0
    checkPLamSystem t0 va ps
    vt  <- evalTyping t0
    rho <- asks env
    let vps = E.evalSystem rho ps
    return (VPathP va vt (E.compLine va vt vps))
  PCon c a es phis -> do
    check VU a
    va <- evalTyping a
    (bs,nu) <- getLblType c va
    checks (bs,nu) es
    mapM_ checkFormula phis
    return va
  IdJ a u c d x p -> do
    check VU a
    va <- evalTyping a
    check va u
    vu <- evalTyping u
```

```
    let refu = VIdPair (constPath vu) $ C.mkSystem [(C.eps,vu)]
    rho <- asks env
    let z = Var "z"
        ctype = E.eval rho $ Pi $ Lam "z" a $ Pi $ Lam "_" (Id a u z) U
    check ctype c
    vc <- evalTyping c
    check (E.app (E.app vc vu) refu) d
    check va x
    vx <- evalTyping x
    check (VId va vu vx) p
    vp <- evalTyping p
    return (E.app (E.app vc vx) vp)
  _ -> throwError ("infer " ++ show e)

-- Not used since we have U : U
--
-- (=?=) :: Typing Ter -> Ter -> Typing ()
-- m =?= s2 = do
--   s1 <- m
--   unless (s1 == s2) $ throwError (show s1 ++ " =/= " ++ show s2)
--
-- checkTs :: [(String,Ter)] -> Typing ()
-- checkTs [] = return ()
-- checkTs ((x,a):xas) = do
--   checkType a
--   local (addType (x,a)) (checkTs xas)
--
-- checkType :: Ter -> Typing ()
-- checkType t = case t of
--   U              -> return ()
--   Pi a (Lam x b) -> do
--     checkType a
--     local (addType (x,a)) (checkType b)
--   _ -> infer t =?= U
::::::::::::::
Exp.cf
::::::::::::::
entrypoints Module, Exp ;

comment "--" ;
comment "{-" "-}" ;

layout "where", "let", "split", "mutual", "with" ;
layout stop "in" ;
-- Do not use layout toplevel as it makes pExp fail!

Module.   Module ::= "module" AIdent "where" "{" [Imp] [Decl] "}" ;

Import.   Imp ::= "import" AIdent ;
separator Imp ";" ;

DeclDef.             Decl ::= AIdent [Tele] ":" Exp "=" ExpWhere ;
DeclData.            Decl ::= "data" AIdent [Tele] "=" [Label] ;
DeclHData.           Decl ::= "hdata" AIdent [Tele] "=" [Label] ;
DeclSplit.           Decl ::= AIdent [Tele] ":" Exp "=" "split" "{" [Branch] "}" ;
DeclUndef.           Decl ::= AIdent [Tele] ":" Exp "=" "undefined" ;
DeclMutual.          Decl ::= "mutual" "{" [Decl] "}" ;
DeclOpaque.          Decl ::= "opaque" AIdent ;
DeclTransparent.     Decl ::= "transparent" AIdent ;
DeclTransparentAll.  Decl ::= "transparent_all" ;
separator            Decl ";" ;

Where.    ExpWhere ::= Exp "where" "{" [Decl] "}" ;
NoWhere.  ExpWhere ::= Exp ;

Let.          Exp  ::= "let" "{" [Decl] "}" "in" Exp ;
Lam.          Exp  ::= "\\" [PTele] "->" Exp ;
PLam.         Exp  ::= "<" [AIdent] ">" Exp ;
Split.        Exp  ::= "split@" Exp "with" "{" [Branch] "}" ;
Fun.          Exp1 ::= Exp2 "->" Exp1 ;
```

```
Pi.           Exp1 ::= [PTele] "->" Exp1 ;
Sigma.        Exp1 ::= [PTele] "*" Exp1 ;
AppFormula.   Exp2 ::= Exp2 "@" Formula ;
App.          Exp2 ::= Exp2 Exp3 ;
PathP.        Exp3 ::= "PathP" Exp4 Exp4 Exp4 ;
Comp.         Exp3 ::= "comp" Exp4 Exp4 System ;
HComp.        Exp3 ::= "hComp" Exp4 Exp4 System ;
Trans.        Exp3 ::= "transport" Exp4 Exp4 ;
Fill.         Exp3 ::= "fill" Exp4 Exp4 System ;
Glue.         Exp3 ::= "Glue" Exp4 System ;
GlueElem.     Exp3 ::= "glue" Exp4 System ;
UnGlueElem.   Exp3 ::= "unglue" Exp4 System ;
Id.           Exp3 ::= "Id" Exp4 Exp4 Exp3 ;
IdPair.       Exp3 ::= "idC" Exp4 System ;
IdJ.          Exp3 ::= "idJ" Exp4 Exp4 Exp4 Exp4 Exp4 Exp4 ;
Fst.          Exp4 ::= Exp4 ".1" ;
Snd.          Exp4 ::= Exp4 ".2" ;
Pair.         Exp5 ::= "(" Exp "," [Exp] ")" ;
Var.          Exp5 ::= AIdent ;
PCon.         Exp5 ::= AIdent "{" Exp "}" ; -- c{T A B} x1 x2 @ phi
U.            Exp5 ::= "U" ;
Hole.         Exp5 ::= HoleIdent ;
coercions Exp 5 ;
separator nonempty Exp "," ;

Dir0.       Dir ::= "0" ;
Dir1.       Dir ::= "1" ;

System.     System ::= "[" [Side] "]" ;

Face.     Face ::= "(" AIdent "=" Dir ")" ;
separator Face "" ;

Side.   Side ::= [Face] "->" Exp ;
separator Side "," ;

Disj.     Formula  ::= Formula "\\/" Formula1 ;
Conj.     Formula1 ::= Formula1 CIdent Formula2 ;
Neg.      Formula2 ::= "-" Formula2 ;
Atom.     Formula2 ::= AIdent ;
Dir.      Formula2 ::= Dir ;
coercions Formula 2 ;

-- Branches
OBranch.   Branch ::= AIdent [AIdent] "->" ExpWhere ;
-- TODO: better have ... @ i @ j @ k -> ... ?
PBranch.   Branch ::= AIdent [AIdent] "@" [AIdent] "->" ExpWhere ;
separator Branch ";" ;

-- Labelled sum alternatives
OLabel.   Label ::= AIdent [Tele] ;
PLabel.   Label ::= AIdent [Tele] "<" [AIdent] ">" System ;
separator Label "|" ;

-- Telescopes
Tele.     Tele ::= "(" AIdent [AIdent] ":" Exp ")" ;
terminator Tele "" ;

-- Nonempty telescopes with Exp:s, this is hack to avoid ambiguities
-- in the grammar when parsing Pi
PTele.    PTele ::= "(" Exp ":" Exp ")" ;
terminator nonempty PTele "" ;

position token AIdent ('_')|(letter)(letter|digit|'\''|'_')*|('!')(digit)* ;
separator AIdent "" ;

token CIdent '/''\\' ;

position token HoleIdent '?' ;
```