```
2 Connections.hs
 3 :::::::::::
 4 -- | Models the lattice of formulas.
 5 {-# LANGUAGE TypeSynonymInstances, FlexibleInstances,
                GeneralizedNewtypeDeriving, TupleSections #-}
7 module Connections where
9 import Control.Applicative
10 import Data.List
11 import Data.Map (Map,(!),keys,fromList,toList,mapKeys,elems,intersectionWith
                   ,unionWith,singleton,foldrWithKey,assocs,mapWithKey
                   , filterWithKey, member)
14 import Data.Set (Set,isProperSubsetOf)
15 import qualified Data. Map as Map
16 import qualified Data. Set as Set
17 import Data.Maybe
18 import Test.QuickCheck
19
20 newtype Name = Name String
    deriving (Arbitrary, Eq, Ord)
21
22
23 instance Show Name where
    show (Name i) = i
24
26 swapName :: Name -> (Name, Name) -> Name
27 swapName k(i,j) \mid k == i
                              = j
28
                    | k == j
29
                    | otherwise = k
30
31 -- | Directions
32 data Dir = Zero | One
    deriving (Eq,0rd)
33
34
35 instance Show Dir where
   show Zero = "0"
36
     show One = "1"
37
38
39 instance Num Dir where
40
   Zero + Zero = Zero
41
               = 0ne
42
     Zero * _ = Zero
43
    One *\bar{x} = x
44
45
46
     abs
            = id
47
     signum_= 0ne
48
49
     negate Zero = One
50
     negate One = Zero
51
52
     fromInteger 0 = Zero
     fromInteger 1 = 0ne
53
     fromInteger _ = error "fromInteger Dir"
54
55
56 instance Arbitrary Dir where
57
    arbitrary = do
58
       b <- arbitrary
59
       return $ if b then Zero else One
60
61 -- | Face
62
63 -- Faces of the form: [(i,0),(j,1),(k,0)]
64 type Face = Map Name Dir
66 instance {-# OVERLAPPING #-} Arbitrary Face where
67
     arbitrary = fromList <$> arbitrary
68
69 showFace :: Face -> String
```

```
70 showFace alpha = concat [ "(" ++ show i ++ " = " ++ show d ++ ")"
71
                            [ (i,d) <- toList alpha ]</pre>
72
73 swapFace :: Face -> (Name, Name) -> Face
74 swapFace alpha ij = mapKeys (`swapName` ij) alpha
75
76 -- Check if two faces are compatible
77 compatible :: Face -> Face -> Bool
78 compatible xs ys = and (elems (intersectionWith (==) xs ys))
79
80 compatibles :: [Face] -> Bool
81 compatibles []
                     = True
82 compatibles (x:xs) = all (x `compatible`) xs && compatibles xs
83
84 allCompatible :: [Face] -> [(Face, Face)]
85 allCompatible [] = []
86 allCompatible (f:fs) = map (f,) (filter (compatible f) fs) ++ allCompatible fs
87
88 -- Partial composition operation
89 meet :: Face -> Face
90 meet = unionWith f
     where f d1 d2 = if d1 == d2 then d1 else error "meet: incompatible faces"
91
93 meetMaybe :: Face -> Face -> Maybe Face
94 meetMaybe x y = if compatible x y then Just $ meet x y else Nothing
96 -- | Quick check test that meet is commutative
97 meetCom :: Face -> Face -> Property
98 meetCom xs ys = compatible xs ys ==> xs `meet` ys == ys `meet` xs
99
100 -- | Quick check test that meet is associative
101 meetAssoc :: Face -> Face -> Property
102 meetAssoc xs ys zs = compatibles [xs,ys,zs] ==>
103
                         xs `meet` (ys `meet` zs) == (xs `meet` ys) `meet` zs
104
105 meetId :: Face -> Bool
106 meetId xs = xs `meet` xs == xs
107
108 -- | all meets of all compatible elements.
109 meets :: [Face] -> [Face] -> [Face]
110 meets xs ys = nub [ meet x y | x <- xs, y <- ys, compatible x y ]
112 meetss :: [[Face]] -> [Face]
113 meetss = foldr meets [eps]
115 -- | a \le b iff a / b = a. Recall that meet is partial.
116 leg :: Face -> Face -> Bool
117 alpha `leq` beta = meetMaybe alpha beta == Just alpha
118
119 comparable :: Face -> Face -> Bool
120 comparable alpha beta = alpha `leq` beta || beta `leq` alpha
121
122 -- | TODO: seems sensitive to ordering?
123 incomparables :: [Face] -> Bool
124 incomparables []
                        = True
125 incomparables (x:xs) = all (not . (x `comparable`)) xs && incomparables xs
126
127 (~>) :: Name -> Dir -> Face
128 i \rightarrow d = singleton i d
129
130 eps :: Face
131 \text{ eps} = Map.empty
132
133 minus :: Face -> Face -> Face
134 minus alpha beta = alpha Map.\\ beta
136 -- Compute the witness of A <= B, ie compute C s.t. B = CA
137 -- leqW :: Face -> Face
138 -- legW = undefined
139
```

```
140 -- | Logical Formulas over this algebra.
141 data Formula = Dir Dir
142
                  | Atom Name
                  | NegAtom Name
143
144
                  | Formula :/\: Formula
145
                  | Formula :\/: Formula
146
      deriving Eq
147
148 instance Show Formula where
      show (Dir Zero) = "0"
149
                        = "1"
150
      show (Dir One)
      show (NegAtom a) = '-' : show a
151
152
      show (Atom a)
                     = show a
      show (a :\/: b) = show1 a ++ " \\/ " ++ show1 b
153
        where show1 v@(a :/\: b) = "(" ++ show v ++ ")"
154
155
               show1 a = show a
      show (a :/\: b) = show1 a ++ " /\\ " ++ show1 b
156
        where show1 v@(a :\/: b) = "(" ++ show v ++ ")"
157
158
               show1 a = show a
159
160 arbFormula :: [Name] -> Int -> Gen Formula
161 arbFormula names s =
      frequency [ (1, Dir <$> arbitrary)
162
                 , (1, Atom <$> elements names)
163
164
                 , (1, NegAtom <$> elements names)
165
                 , (s, do op <- elements [andFormula,orFormula]</pre>
166
                          op <$> arbFormula names s' <*> arbFormula names s')
167
      where s' = s `div` 3
168
169
170 instance Arbitrary Formula where
      arbitrary = do
171
172
          n <- arbitrary :: Gen Integer</pre>
173
          sized $ arbFormula (map (\x ->  Name ('!' : show x)) [0..(abs n)])
174
175 class ToFormula a where
176
      toFormula :: a -> Formula
177
178 instance ToFormula Formula where
179
     toFormula = id
180
181 instance ToFormula Name where
    toFormula = Atom
182
183
184 instance ToFormula Dir where
185
     toFormula = Dir
186
187 negFormula :: Formula -> Formula
                          = Dir (- b) -- exploit Num instance of Dir.
188 negFormula (Dir b)
189 negFormula (Atom i)
                               = NegAtom i
190 negFormula (NegAtom i)
                             = Atom i
191 -- | use smart constructors orFormula, andFormula for constant folding.
192 negFormula (phi :/\: psi) = orFormula (negFormula phi) (negFormula psi)
193 negFormula (phi :\/: psi) = andFormula (negFormula phi) (negFormula psi)
194
195 andFormula :: Formula -> Formula -> Formula
196 andFormula (Dir Zero) _ = Dir Zero
197 andFormula _ (Dir Zero) = Dir Zero
198 andFormula (Dir One) phi = phi
199 andFormula phi (Dir One) = phi
200 andFormula phi psi
                               = phi :/\: psi
201
202 orFormula :: Formula -> Formula -> Formula
203 orFormula (Dir One)
                              = Dir One
204 orFormula _ (Dir One) = Dir
205 orFormula (Dir Zero) phi = phi
                              = Dir One
206 orFormula phi (Dir Zero) = phi
                               = phi :\/: psi
207 orFormula phi psi
208
209 -- | formula to sets of solutions (?)
```

```
210 dnf :: Formula -> Set (Set (Name, Dir))
211 dnf (Dir One)
                      = Set.singleton Set.empty
212 dnf (Dir Zero)
                       = Set.empty
213 dnf (Atom n)
                       = Set.singleton (Set.singleton (n,1))
214 dnf (NegAtom n)
                       = Set.singleton (Set.singleton (n,0))
215 dnf (phi :\/: psi) = dnf phi `merge` dnf psi
216 dnf (phi :/\: psi) =
217
      foldr merge Set.empty [ Set.singleton (a `Set.union` b)
218
                             | a <- Set.toList (dnf phi)
219
                             , b <- Set.toList (dnf psi) ]</pre>
220
221 fromDNF :: Set (Set (Name, Dir)) -> Formula
222 fromDNF s = foldr (orFormula . foldr andFormula (Dir One)) (Dir Zero) fs
      where xss = map Set.toList $ Set.toList s
223
224
            fs = [ [if d == Zero then NegAtom n else Atom n | (n,d) <- xs ] | xs <- xss ]
225
226 merge :: Set (Set (Name,Dir)) -> Set (Set (Name,Dir)) -> Set (Set (Name,Dir))
227 merge ab =
      let as = Set.toList a
228
229
          bs = Set.toList b
230
      in Set.fromList [ ai | ai <- as, not (any (`isProperSubsetOf` ai) bs) ] `Set.union`</pre>
         Set.fromList [ bi | bi <- bs, not (any (`isProperSubsetOf` bi) as) ]</pre>
231
232
233 -- evalFormula :: Formula -> Face -> Formula
234 -- evalFormula phi alpha =
235 --
        Map.foldWithKey (\i d psi -> act psi (i,Dir d)) phi alpha
236
237
    -- (Dir b) alpha = Dir b
238 -- evalFormula (Atom i) alpha = case Map.lookup i alpha of
239 --
                                       Just b -> Dir b
240 --
                                       Nothing -> Atom i
241 -- evalFormula (Not phi) alpha = negFormula (evalFormula phi alpha)
242 -- evalFormula (phi :/\: psi) alpha =
        andFormula (evalFormula phi alpha) (evalFormula psi alpha)
244 -- evalFormula (phi :\/: psi) alpha =
245 --
        orFormula (evalFormula phi alpha) (evalFormula psi alpha)
246
247 -- TODO: think about what the hell this means.
248 -- find a better name?
249 -- phi b = max {alpha : Face | phi alpha = b}
250 invFormula :: Formula -> Dir -> [Face]
251 invFormula (Dir b') b
                                   = [ eps | b == b' ]
252 invFormula (Atom i) b
                                   = [ singleton i b ]
253 invFormula (NegAtom i) b
                                   = [ singleton i (-b) ]
254 invFormula (phi :/\: psi) Zero = invFormula phi 0 `union` invFormula psi 0
255 invFormula (phi :/\: psi) One = meets (invFormula phi 1) (invFormula psi 1)
256 invFormula (phi :\/: psi) b
                                   = invFormula (negFormula phi :/\: negFormula psi) (- b)
258 propInvFormulaIncomp :: Formula -> Dir -> Bool
259 propInvFormulaIncomp phi b = incomparables (invFormula phi b)
260
261 -- prop invFormula :: Formula -> Dir -> Bool
262 -- prop_invFormula phi b =
263 --
        all (\alpha -> phi `evalFormula` alpha == Dir b) (invFormula phi b)
264
265 -- testInvFormula :: [Face]
266 -- testInvFormula = invFormula (Atom (Name 0) :/\: Atom (Name 1)) 1
267
268 -- | Nominal
269
270 -- gensym :: [Name] -> Name
271 -- gensym xs = head (ys \mid \mid xs)
         where ys = map \; Name \; \{ "i", "j", "k", "l" \} ++ map \; (('i':) . show) \; [0..]
272 --
273
274 -- gensymNice :: Name -> [Name] -> Name
275 -- gensymNice i@(Name\ s)\ xs = head\ (ys\ \ xs)
276 --
         where ys = i:map(n -> Name(s ++ show n))[0..]
278 -- | Assumes names are named \cdot!x.
279 gensym :: [Name] -> Name
```

```
280 gensym xs = Name ('!' : show max)
      where max = maximum' [ read x | Name ('!':x) \leftarrow xs ]
281
282
            maximum'[] = 0
283
            maximum' xs = maximum xs + 1
284
285 gensyms :: [Name] -> [Name]
286 gensyms d = let x = gensym d in x : gensyms (x : d)
287
288 class Nominal a where
289
      -- | return all names in play. aka support of the
290
      -- partial function [name -> value]
291
      support :: a -> [Name]
292
      -- | a[Name/Formula]. Substitute Name := Formula into a.
293
      act
             :: a -> (Name, Formula) -> a
      -- | swap a (x, y) will swap occurences x <-> y.
294
295
            :: a -> (Name, Name) -> a
      swap
296
297 fresh :: Nominal a => a -> Name
298 fresh = gensym . support
300 -- freshNice :: Nominal a => Name -> a -> Name
301 -- freshNice i = gensymNice i . support
303 freshs :: Nominal a => a -> [Name]
304 freshs = gensyms . support
305
306 unions :: Eq a => [[a]] -> [a]
307 unions = foldr union []
308
309 unionsMap :: Eq b => (a -> [b]) -> [a] -> [b]
310 unionsMap f = unions . map f
311
312 newtype Nameless a = Nameless { unNameless :: a }
313
                       deriving (Eq, Ord)
314
315 instance Nominal (Nameless a) where
316
      support _ = []
      act x _
317
318
      swap x _ = x
319
320 instance Nominal () where
321
      support () = []
      act () _ = ()
322
      swap ()__
323
                = ()
324
325 instance (Nominal a, Nominal b) => Nominal (a, b) where
326
      support (a, b) = support a `union` support b
327
      act (a,b) f
                     = (act a f,act b f)
328
                     = (swap a n,swap b n)
      swap (a,b) n
329
330 instance (Nominal a, Nominal b, Nominal c) => Nominal (a, b, c) where
331
      support (a,b,c) = unions [support a, support b, support c]
332
      act (a,b,c) f = (act a f,act b f,act c f)
333
      swap (a,b,c) n = (swap a n,swap b n,swap c n)
334
335 instance (Nominal a, Nominal b, Nominal c, Nominal d) =>
336
             Nominal (a, b, c, d) where
337
      support (a,b,c,d) = unions [support a, support b, support c, support d]
338
      act (a,b,c,d) f = (act a f,act b f,act c f,act d f)
339
      swap (a,b,c,d) n = (swap a n,swap b n,swap c n,swap d n)
340
341 instance (Nominal a, Nominal b, Nominal c, Nominal d, Nominal e) =>
342
             Nominal (a, b, c, d, e) where
343
      support (a,b,c,d,e)
344
        unions [support a, support b, support c, support d, support e]
345
                           = (act a f,act b f,act c f,act d f, act e f)
      act (a,b,c,d,e) f
346
      swap (a,b,c,d,e) n =
347
        (swap a n,swap b n,swap c n,swap d n,swap e n)
348
349 instance (Nominal a, Nominal b, Nominal c, Nominal d, Nominal e, Nominal h) =>
```

```
350
             Nominal (a, b, c, d, e, h) where
351
      support (a,b,c,d,e,h) =
352
        unions [support a, support b, support c, support d, support e, support h]
353
      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)
354
      swap (a,b,c,d,e,h) n =
355
        (swap a n,swap b n,swap c n,swap d n,swap e n,swap h n)
356
357 instance Nominal a => Nominal [a] where
358
      support xs = unions (map support xs)
359
                 = [act x f | x <- xs <math>]
      act xs f
                  = [ swap x n | x < - xs ]
360
      swap xs n
361
362 instance Nominal a => Nominal (Maybe a) where
363
      support
               = maybe [] support
                 = fmap (`act` f) v
364
      act v f
      swap a n = fmap (`swap` n) a
365
366
367 instance Nominal Formula where
      support (Dir _)
                             = []
368
                             = [i]
369
      support (Atom i)
                           = [i]
370
      support (NegAtom i)
      support (phi :/\: psi) = support phi `union` support psi
371
      support (phi :\/: psi) = support phi `union` support psi
372
373
374
      act (Dir b) (i,phi) = Dir b
375
      act (Atom j) (i,phi) | i == j
                                       = phi
376
                           | otherwise = Atom j
      act (NegAtom j) (i,phi) | i == j = negFormula phi
377
378
                              | otherwise = NegAtom j
379
      act (psi1 :/\: psi2) (i,phi) = act psi1 (i,phi) `andFormula` act psi2 (i,phi)
380
      act (psi1 :\/: psi2) (i,phi) = act psi1 (i,phi) `orFormula` act psi2 (i,phi)
381
382
      swap (Dir b) (i,j) = Dir b
383
      swap (Atom k) (i,j) \mid k == i
                                     = Atom j
                                    = Atom i
384
                         | k == j
                         | otherwise = Atom k
385
      swap (NegAtom k) (i,j) \mid k == i
386
                                      = NegAtom j
                                       = NegAtom i
387
                            | k == j
388
                             | otherwise = NegAtom k
      swap (psil :/\: psi2) (i,j) = swap psil (i,j) :/\: swap psi2 (i,j)
389
      swap (psi1 : \ \ (i,j) = swap psi1 (i,j) : \ \ swap psi2 (i,j)
390
391
392 face :: Nominal a => a -> Face -> a
393 face = foldrWithKey (\i d a -> act a (i,Dir d))
395 -- | the faces should be incomparable.
396 -- | "partial components" of a n-cube.
397 type System a = Map Face a
398
399 showListSystem :: Show a => [(Face,a)] -> String
400 \text{ showListSystem } [] = "[]"
401 showListSystem ts =
      "[ " ++ intercalate ", " [ showFace alpha ++ " -> " ++ show u
402
                               | (alpha,u) <- ts ] ++ " ]"
403
404
405 showSystem :: Show a => System a -> String
406 showSystem = showListSystem . toList
408 -- | If face is <= any key, don't insert.
409 -- | Otherwise, insert face and remove all those faces which are <= this face.
410 -- | ie, can only "inflate" the system by adding larger keys, which then
411 -- | enforce their compatibility condition.
412 -- TODO: what about >= ?
413 insertSystem :: Face -> a -> System a -> System a
414 insertSystem alpha v ts
415
     | any (leq alpha) (keys ts) = ts
416
      | otherwise = Map.insert alpha v
417
                    (Map.filterWithKey (\gamma _ -> not (gamma `leq` alpha)) ts)
418
419 insertsSystem :: [(Face, a)] -> System a -> System a
```

```
420 insertsSystem faces us = foldr (uncurry insertSystem) us faces
421
422 mkSystem :: [(Face, a)] -> System a
423 mkSystem = flip insertsSystem Map.empty
424
425 unionSystem :: System a -> System a -> System a
426 unionSystem us vs = insertsSystem (assocs us) vs
427
428
429 -- | Monad?! WTF.
430 joinSystem :: System (System a) -> System a
431 joinSystem tss = mkSystem $
      [ (alpha `meet` beta,t) | (alpha,ts) <- assocs tss, (beta,t) <- assocs ts ]
433
434
435 -- TODO: add some checks
436 transposeSystemAndList :: System [a] -> [b] -> [(System a,b)]
437 transposeSystemAndList
                            _ []
438 transposeSystemAndList tss (u:us) =
439
      (Map.map head tss,u):transposeSystemAndList (Map.map tail tss) us
441 -- Quickcheck this:
442 -- (i = phi) * beta = (beta - i) * (i = phi beta)
444 -- Now we ensure that the keys are incomparable
445 instance Nominal a => Nominal (System a) where
      support s = unions (map keys $ keys s)
447
                  `union` support (elems s)
448
449
      act s (i, phi) = addAssocs (assocs s)
450
        where
451
        addAssocs [] = Map.empty
452
        addAssocs ((alpha,u):alphaus) =
453
          let s' = addAssocs alphaus
454
          in case Map.lookup i alpha of
455
            Just d -> let beta = Map.delete i alpha
456
                      in foldr (\delta s'' -> insertSystem (meet delta beta)
                                                 (face u (Map.delete i delta)) s'')
457
458
                                                 s' (invFormula (face phi beta) d)
            Nothing -> insertSystem alpha (act u (i,face phi alpha)) s'
459
460
461
      swap s ij = mapKeys (`swapFace` ij) (Map.map (`swap` ij) s)
462
463 -- | TODO bollu: is border . border = 0? :)
464 -- | carve a using the same shape as the system b
465 border :: Nominal a => a -> System b -> System a
466 border v = mapWithKey (const . face v)
468 shape :: System a -> System ()
469 shape = border ()
470
471 instance {-# OVERLAPPING #-} (Nominal a, Arbitrary a) => Arbitrary (System a) where
472
      arbitrary = do
473
        a <- arbitrary
474
        border a <$> arbitraryShape (support a)
475
476
          arbitraryShape :: [Name] -> Gen (System ())
477
          arbitraryShape supp = do
478
            phi <- sized $ arbFormula supp</pre>
479
            return $ fromList [(face,()) | face <- invFormula phi 0]
480
481 sym :: Nominal a => a -> Name -> a
482 sym a i = a `act` (i, NegAtom i)
483
484 rename :: Nominal a => a -> (Name, Name) -> a
485 rename a (i, j) = a \cdot act \cdot (i, Atom j)
486
487 conj, disj :: Nominal a => a -> (Name, Name) -> a
488 conj a (i, j) = a act (i, Atom i :/\: Atom j)
489 disj a (i, j) = a `act` (i, Atom i :\/: Atom j)
```

```
490
491 legSystem :: Face -> System a -> Bool
492 alpha `leqSystem` us =
493
     not $ Map.null $ filterWithKey (\beta _ -> alpha `leq` beta) us
494
495 -- assumes alpha <= shape us
496 proj :: (Nominal a, Show a) => System a -> Face -> a
497 proj us alpha | eps `member` usalpha = usalpha ! eps
498
                  | otherwise
      error $ "proj: eps not in " ++ show usalpha ++ "\nwhich is the "
499
        ++ show alpha ++ "\nface of " ++ show us
500
      where usalpha = us `face` alpha
501
502
503 domain :: System a -> [Name]
504 domain = keys . Map.unions . keys
505 ::::::::::::
506 CTT.hs
507 :::::::::::::
508 {-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}
509 module CTT where
511 import Control.Applicative
512 import Data.List
513 import Data. Maybe
514 import Data.Map (Map,(!),filterWithKey,elems)
515 import qualified Data. Map as Map
516 import Text.PrettyPrint as PP
517 import Data.Set (Set)
518 import qualified Data. Set as Set
519 import Prelude hiding ((<>))
520
521 import qualified Connections as C
522
523 ----
524 -- | Terms, with type |Ter|.
525 -- | General conventions: O for object, P for path.
526
527
528 -- | File locations
529 data Loc = Loc { locFile :: String
                   , locPos :: (Int,Int) }
530
531
     deriving Eq
532
533 type Ident = String
534 -- | TODO: Identifier of Telescopes. Are of two types, Object and path based.
535 type LIdent = String
536
537 -- Telescope (x1 : A1) .. (xn : An)
538 type Tele = [(Ident,Ter)]
539
540 -- | from Exp.cf:
                        C.System ::= "[" [Side] "]" ;|
541 --
          C.System.
           Side. Side ::= [Face] "->" Exp ;
542 --
543 -- |
           separator Side ",";
544 -- |
                    Face ::= "(" AIdent "=" Dir ")" ;
          Face.
          separator Face "" ;
545 -- |
546 -- | C.System comes from Connections.hs.
547 -- | C.System Ter is a map from Face to Term,
548 --
            where the faces are incomparable.
549 data Label = OLabel LIdent Tele -- Object label
550
               | PLabel LIdent Tele [C.Name] (C.System Ter) -- Path label
551
     deriving (Eq,Show)
552
553 -- \mid OBranch of the form: c \times 1 \dots \times n -> e
554 -- | PBranch of the form: c x1 .. xn i1 .. im -> e
555 data Branch = OBranch LIdent [Ident] Ter
                | PBranch LIdent [Ident] [C.Name] Ter
556
557
      deriving (Eq,Show)
558
559 -- Declarations: x : A = e
```

```
560 -- A group of mutual declarations is identified by its location. It is used to
561 -- speed up the Eq instance for Ctxt.
562 type Decl = (Ident,(Ter,Ter))
563 data Decls = MutualDecls Loc [Decl]
564
               | OpaqueDecl Ident
               | TransparentDecl Ident
565
566
               | TransparentAllDecl
567
               deriving Eq
568
569 declIdents :: [Decl] -> [Ident]
570 declIdents decls = [x \mid (x, ) < - decls]
572 declTers :: [Decl] -> [Ter]
573 declTers decls = [d \mid (\_,(\_,d)) < - decls]
574
575 -- | convert a sequence of declarations into a sequence of (ident: type)
576 declTele :: [Decl] -> Tele
577 declTele decls = [(x,t) | (x,(t,_)) < - decls]
579 declDefs :: [Decl] -> [(Ident,Ter)]
580 declDefs decls = [(x,d) | (x,(_,d)) < - decls]
581
582 labelTele :: Label -> (LIdent, Tele)
583 labelTele (OLabel c ts) = (c,ts)
584 labelTele (PLabel c ts _ _) = (c,ts)
586 labelName :: Label -> LIdent
587 labelName = fst . labelTele
588
589 labelTeles :: [Label] -> [(LIdent,Tele)]
590 labelTeles = map labelTele
592 lookupLabel :: LIdent -> [Label] -> Maybe Tele
593 lookupLabel x xs = lookup x (labelTeles xs)
595 lookupPLabel :: LIdent -> [Label] -> Maybe (Tele,[C.Name],C.System Ter)
596 lookupPLabel x xs = listToMaybe [ (ts,is,es) | PLabel y ts is es \leftarrow xs, x == y ]
597
598 branchName :: Branch -> LIdent
599 branchName (OBranch c \_ \_) = c
600 branchName (PBranch c \_ \_ ) = c
601
602 lookupBranch :: LIdent -> [Branch] -> Maybe Branch
603 lookupBranch _ [] = Nothing
604 lookupBranch x (b:brs) = case b of
      OBranch \ c \ \_ \ | \ x == c \ -> Just \ b
605
606
                      | otherwise -> lookupBranch x brs
607
      PBranch c \_ \_ | x == c \longrightarrow Just b
608
                       | otherwise -> lookupBranch x brs
609
610 -- TODO: Term v/s Value?
611 -- Terms
612 data Ter = Pi Ter -- TODO: ?
             | App Ter Ter -- f x
613
             | Lam Ident Ter Ter -- \x: T. e
614
             | Where Ter Decls -- TODO: ?
615
             | Var Ident -- x
616
617
             | U -- Unit
618
               -- Sigma types:
             | Sigma Ter -- TODO: ?
619
             | Pair Ter Ter -- (a, b)
620
             | Fst Ter -- fst t
621
             | Snd Ter -- snd t
622
623
               -- constructor c Ms
624
             | Con LIdent [Ter]
             PCon LIdent Ter [Ter] [C.Formula] -- c A ts phis (A is the data type)
625
626
               -- branches c1 xs1 -> M1,..., cn xsn -> Mn
             | Split Ident Loc Ter [Branch]
627
               -- labelled sum c1 Als,..., cn Ans (assumes terms are constructors)
628
             | Sum Loc Ident [Label] -- TODO: should only contain OLabels
629
```

```
630
             | HSum Loc Ident [Label]
631
              -- undefined and holes
632
             | Undef Loc Ter -- Location and type
633
             | Hole Loc
              -- Path types
634
635
             | PathP Ter Ter Ter
             | PLam C.Name Ter
636
             | AppFormula Ter C.Formula
637
               -- Kan composition and filling
638
             | Comp Ter Ter (C.System Ter)
| Fill Ter Ter (C.System Ter)
639
640
641
             | HComp Ter Ter (C.System Ter)
642
             -- Glue
             | Glue Ter (C.System Ter)
643
             | GlueElem Ter (C.System Ter)
644
645
             | UnGlueElem Ter (C.System Ter)
646
              -- Id
647
             | Id Ter Ter Ter
             i IdPair Ter (C.System Ter)
648
649
             | IdJ Ter Ter Ter Ter Ter
650
     deriving Eq
651
652 -- For an expression t, returns (u,ts) where u is no application and t=u ts
653 unApps :: Ter -> (Ter,[Ter])
654 \text{ unApps} = \text{aux} []
     where aux :: [Ter] -> Ter -> (Ter,[Ter])
656
            aux acc (App r s) = aux (s:acc) r
657
                             = (t,acc)
            aux acc t
658
659 mkApps :: Ter -> [Ter] -> Ter
660 mkApps (Con l us) vs = Con l (us ++ vs)
661 mkApps t ts
                        = foldl App t ts
662
663 mkWheres :: [Decls] -> Ter -> Ter
664 mkWheres []
                e = e
665 mkWheres (d:ds) e = Where (mkWheres ds e) d
667 --------
668 -- | Values
669
670 data Val = VU
             | Ter Ter Env
671
             I VPi Val Val
672
             | VSigma Val Val
673
             | VPair Val Val
674
675
             | VCon LIdent [Val]
             | VPCon LIdent Val [Val] [C.Formula]
676
677
678
               -- Path values
679
             | VPathP Val Val Val
680
             | VPLam C.Name Val
681
             | VComp Val Val (C.System Val)
682
683
               -- Glue values
             | VGlue Val (C.System Val)
684
685
             | VGlueElem Val (C.System Val)
686
             | VUnGlueElem Val (C.System Val)
687
688
               -- Composition in the universe
689
             | VCompU Val (C.System Val)
690
691
               -- Composition for HITs; the type is constant
692
             | VHComp Val Val (C.System Val)
693
               -- Id
694
695
             | VId Val Val Val
             | VIdPair Val (C.System Val)
696
697
698
               -- TODO: Neutral => normalization by evaluation?
699
               -- Neutral values:
```

```
700
                            | VVar Ident Val
701
                             | VOpaque Ident Val
702
                             | VFst Val
                             | VSnd Val
703
704
                             | VSplit Val Val
705
                             | VApp Val Val
706
                             | VAppFormula Val C.Formula
707
                               VLam Ident Val Val
708
                             | VUnGlueElemU Val Val (C.System Val)
709
                             I VIdJ Val Val Val Val Val Val
710
            deriving Eq
711
712 isNeutral :: Val -> Bool
713 isNeutral v = case \ v \ of
             Ter Undef{} _ -> True
714
                                             -> True
715
             Ter Hole{} _
                                             -> True
716
             VVar{}
                                            -> True
717
             V0paque{}
                                             -> True
718
             VComp{}
                                             -> True
719
             VFst{}
                                             -> True
720
             VSnd{}
                                             -> True
721
             VSplit{}
722
             VApp{}
                                             -> True
723
             VAppFormula{} -> True
             VUnGlueElemU{} -> True
724
             VUnGlueElem{} -> True
725
726
                                             -> True
             VIdJ{}
727
                                             -> False
728
729 isNeutralSystem :: C.System Val -> Bool
730 isNeutralSystem = any isNeutral . elems
731
732 -- isNeutralPath :: Val -> Bool
733 -- isNeutralPath (VPath _ v) = isNeutral v
734 -- isNeutralPath _ = True
736 mkVar :: Int -> String -> Val -> Val
737 mkVar k x = VVar (x ++ show k)
738
739 mkVarNice :: [String] -> String -> Val -> Val
740 mkVarNice xs x = VVar (head (ys \xspace xspace xspac
741
            where ys = x:map (\n -> x ++ show n) [0..]
742
743 unCon :: Val -> [Val]
744 unCon (VCon _ vs) = vs
745 unCon v
                                             = error $ "unCon: not a constructor: " ++ show v
746
747 isCon :: Val -> Bool
748 isCon VCon{} = True
749 isCon _
                                 = False
750
751 -- Constant path: <_> v
752 constPath :: Val -> Val
753 constPath = VPLam (C.Name "_")
754
755
756 ------
757 -- | Environments
758
759 data Ctxt = Empty
760
                              | Upd Ident Ctxt
761
                              | Sub C.Name Ctxt
                              | Def Loc [Decl] Ctxt
762
            deriving (Show)
763
764
765 instance Eq Ctxt where
766 c == d = case(c, d) of
                                                                                      -> True
767
                          (Empty, Empty)
                          (Upd x c', Upd y d')
(Sub i c', Sub j d')
                                                                                   -> x == y && c' == d'
768
                                                                                      -> i == j && c' == d'
769
```

```
(Def m xs c', Def n ys d') \rightarrow (m == n || xs == ys) && c' == d'
 770
 771
                 -- Invariant: if two declaration groups come from the same
 772
                 -- location, they are equal and their contents are not compared.
 773
                                           -> False
 774
 775 -- The Idents and Names in the Ctxt refer to the elements in the two
 776 -- lists. This is more efficient because acting on an environment now
 777 -- only need to affect the lists and not the whole context.
 778 -- The last list is the list of opaque names
 779 -- | C.Nameless comes from Connections.hs
 780 newtype Env = Env (Ctxt,[Val],[C.Formula],C.Nameless (Set Ident))
 781
       deriving (Eq)
 782
 783 emptyEnv :: Env
 784 emptyEnv = Env (Empty,[],[],C.Nameless Set.empty)
 785
 786 def :: Decls -> Env -> Env
 787 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)))
788 def (OpaqueDecl n) (Env (rho, vs, fs, C. Nameless os)) = Env (rho, vs, fs, C. Nameless
(Set.insert n os))
 789 def (TransparentDecl n) (Env (rho,vs,fs,C.Nameless os)) = Env (rho,vs,fs,C.Nameless
(Set.delete n os))
 790 def TransparentAllDecl (Env (rho, vs, fs, C. Nameless os)) = Env (rho, vs, fs, C. Nameless
791
 792 defWhere :: Decls -> Env -> Env
 793 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)))
 794 defWhere (OpaqueDecl _) rho = rho
 795 defWhere (TransparentDecl _) rho = rho
 796 defWhere TransparentAllDec\overline{l} rho = rho
 798 sub :: (C.Name, C.Formula) -> Env -> Env
 799 sub (i,phi) (Env (rho,vs,fs,os)) = Env (Sub i rho,vs,phi:fs,os)
 801 upd :: (Ident, Val) -> Env -> Env
 802 upd (x,v) (Env (rho,vs,fs,C.Nameless os)) = Env <math>(Upd x rho,v:vs,fs,C.Nameless (Set.delete
x os))
 803
 804 upds :: [(Ident, Val)] -> Env -> Env
 805 upds xus rho = foldl (flip upd) rho xus
 807 updsTele :: Tele -> [Val] -> Env -> Env
 808 updsTele tele vs = upds (zip (map fst tele) vs)
 810 subs :: [(C.Name, C.Formula)] -> Env -> Env
 811 subs iphis rho = foldl (flip sub) rho iphis
 812
 813 mapEnv :: (Val -> Val) -> (C.Formula -> C.Formula) -> Env -> Env
 814 mapEnv f g (Env (rho,vs,fs,os)) = Env (rho,map f vs,map g fs,os)
 815
 816 valAndFormulaOfEnv :: Env -> ([Val],[C.Formula])
 817 valAndFormulaOfEnv (Env (\_, vs, fs, \_)) = (vs, fs)
 818
 819 valOfEnv :: Env -> [Val]
 820 valOfEnv = fst . valAndFormulaOfEnv
 821
 822 formulaOfEnv :: Env -> [C.Formula]
 823 formulaOfEnv = snd . valAndFormulaOfEnv
 824
 825 domainEnv :: Env -> [C.Name]
 826 domainEnv (Env (rho,_,_,_)) = domCtxt rho
 827
       where domCtxt rho = case rho of
 828
               Empty
                           ->[]
 829
               Upd _ e
Def _ t:
                           -> domCtxt e
 830
                    ts e -> domCtxt e
               Sub i e
 831
                           -> i : domCtxt e
 832
 833 -- | Extract the context from the environment, used when printing holes
```

```
834 contextOfEnv :: Env -> [String]
835 contextOfEnv rho = case rho of
                                     ->[]
     Env (Empty,_,_,_)
     Env (Upd x e, VVar n t:vs, fs, os) -> (n ++ " : " ++ show t) : contextOfEnv (Env)
837
(e,vs,fs,os))
838 Env (Upd x e,v:vs,fs,os)
                                     -> (x ++ " = " ++ show v) : context0fEnv (Env
(e, vs, fs, os))
                 _ e,vs,fs,os)
839
     Env (Def _
                                     -> context0fEnv (Env (e,vs,fs,os))
     Env (Sub i e,vs,phi:fs,os)
                                     -> (show i ++ " = " ++ show phi) : contextOfEnv (Env
(e, vs, fs, os))
841
842 -----
843 -- | Pretty printing
844
845 instance Show Env where
     show = render . showEnv True
846
847
848 showEnv :: Bool -> Env -> Doc
849 showEnv b e =
      let -- This decides if we should print "x = " or not
850
851
          names x = if b then text x <+> equals else PP.empty
          par x = if b then parens x else x
852
853
                = if b then comma else PP.empty
854
          showEnv1 e = case e of
855
            Env (Upd x env,u:us,fs,os)
856
              showEnv1 (Env (env,us,fs,os)) <+> names x <+> showVal1 u <> com
            Env (Sub i env,us,phi:fs,os) ->
857
              showEnv1 (Env (env,us,fs,os)) <+> names (show i) <+> text (show phi) <> com
858
859
            Env (Def _ _ env,vs,fs,os) -> showEnv1 (Env (env,vs,fs,os))
860
                                        -> showEnv b e
861
      in case e of
        Env (Empty,_,_,_)
862
                                    -> PP.empty
                                   -> showEnv b (Env (env,vs,fs,os))
863
        Env (Def _ _ env,vs,fs,os)
864
        Env (Upd x env,u:us,fs,os)
865
          par $ showEnv1 (Env (env,us,fs,os)) <+> names x <+> showVal1 u
866
        Env (Sub i env,us,phi:fs,os) ->
867
          par $ showEnv1 (Env (env,us,fs,os)) <+> names (show i) <+> text (show phi)
868
869 instance Show Loc where
     show = render . showLoc
870
871
872 showLoc :: Loc -> Doc
873 showLoc (Loc name (i,j)) = text (show (i,j) ++ " in " ++ name)
874
875 showFormula :: C.Formula -> Doc
876 showFormula phi = case phi of
     _ C.:\/: _ -> parens (text (show phi))
      _ C.:/\: _ -> parens (text (show phi))
878
      _ -> text $ show phi
879
880
881 instance Show Ter where
882
     show = render . showTer
883
884 showTer :: Ter -> Doc
885 showTer v = case \ v \ of
886
                         -> char 'U'
      Ш
      App e0 e1
887
                        -> showTer e0 <+> showTer1 e1
888
      Pi e0
                         -> text "Pi" <+> showTer e0
                        -> char '\\' <> parens (text x <+> colon <+> showTer t) <+>
889
      Lam x t e
890
                             text "->" <+> showTer e
891
      Fst e
                         -> showTer1 e <> text ".1'
                        -> showTer1 e <> text ".2"
892
      Snd e
                        -> text "Sigma" <+> showTer1 e0
893
      Sigma e0
894
      Pair e0 e1
                        -> parens (showTer e0 <> comma <> showTer e1)
                        -> showTer e <+> text "where" <+> showDecls d
895
      Where e d
896
      Var x
                        -> text x
                        -> text c <+> showTers es
897
      Con c es
898
      PCon c a es phis -> text c <+> braces (showTer a) <+> showTers es
899
                         <+> hsep (map ((char '@' <+>) . showFormula) phis)
      Split f _ _ _
                       -> text f
900
```

```
901
      Sum _ n _
                         -> text n
      HSum _ n _
902
                         -> text n
                         -> text "undefined"
903
      Undef{}
                         -> text "?"
904
      Hole{}
                        -> text "PathP" <+> showTers [e0,e1,e2]
905
      PathP e0 e1 e2
      PLam i e
906
                         -> char '<' <> text (show i) <> char '>' <+> showTer e
      AppFormula e phi -> showTer1 e <+> char '@' <+> showFormula phi
907
      Comp e t ts
                         -> text "comp" <+> showTers [e,t] <+> text (C.showSystem ts)
908
                        -> text "hComp" <+> showTers [e,t] <+> text (C.showSystem ts)
909
      HComp e t ts
                         -> text "fill" <+> showTers [e,t] <+> text (C.showSystem ts)
910
      Fill e t ts
                         -> text "Glue" <+> showTer1 a <+> text (C.showSystem ts)
911
      Glue a ts
                         -> text "glue" <+> showTer1 a <+> text (C.showSystem ts)
912
      GlueElem a ts
                         -> text "unglue" <+> showTer1 a <+> text (C.showSystem ts)
913
      UnGlueElem a ts
                         -> text "Id" <+> showTers [a,u,v]
914
      Idauv
                         -> text "idC" <+> showTer1 b <+> text (C.showSystem ts)
      IdPair b ts
915
                       -> text "idJ" <+> showTers [a,t,c,d,x,p]
916
      IdJatcdxp
917
918 showTers :: [Ter] -> Doc
919 showTers = hsep . map showTer1
920
921 showTer1 :: Ter -> Doc
922 showTer1 t = case t of
923
               -> char 'U'
924
      Con c [] -> text c
925
      Var{}
               -> showTer t
926
      Undef{} -> showTer t
927
      Hole{}
               -> showTer t
928
      Split{} -> showTer t
929
              -> showTer t
      Sum{}
930
     HSum{}
              -> showTer t
931
               -> showTer t
      Fst{}
932
      Snd{}
               -> showTer t
933
               -> parens (showTer t)
934
935 showDecls :: Decls -> Doc
936 showDecls (MutualDecls _ defs) =
      hsep $ punctuate comma
      [ text x <+> equals <+> showTer d | (x,(\_,d)) <- defs ]
938
939 showDecls (OpaqueDecl i) = text "opaque" <+> text i
940 showDecls (TransparentDecl i) = text "transparent" <+> text i
941 showDecls TransparentAllDecl = text "transparent_all"
942
943 instance Show Val where
944
     show = render . showVal
945
946 showVal :: Val -> Doc
947 showVal v = case v of
948
      VU
                        -> char 'U'
949
      Ter t@Sum{} rho
                        -> showTer t <+> showEnv False rho
950
      Ter t@HSum{} rho -> showTer t <+> showEnv False rho
951
      Ter t@Split{} rho -> showTer t <+> showEnv False rho
952
      Ter t rho
                        -> showTer1 t <+> showEnv True rho
      VCon c us
953
                        -> text c <+> showVals us
954
      VPCon c a us phis -> text c <+> braces (showVal a) <+> showVals us
955
                           <+> hsep (map ((char '@' <+>) . showFormula) phis)
956
                       -> text "hComp" <+> showVals [v0,v1] <+> text (C.showSystem vs)
      VHComp v0 v1 vs
      VPi a l@(VLam x t b)
    | "_" `isPrefixOf`
957
                           x -> showVal1 a <+> text "->" <+> showVal1 b
958
                             -> char '(' <> showLam v
959
        I otherwise
                        -> text "Pi" <+> showVals [a,b]
      VPi a b
960
961
      VPair u v
                        -> parens (showVal u <> comma <> showVal v)
                        -> text "Sigma" <+> showVals [u,v]
962
      VSigma u v
963
      VApp u v
                        -> showVal u <+> showVal1 v
                        -> text "\\(" <> showLam v
964
      VLam{}
                        -> char '<' <> showPLam v
965
      VPLam{}
966
      VSplit u v
                        -> showVal u <+> showVal1 v
                        -> text x
967
      VVar x _
      V0paque x _
                        -> text ('#':x)
968
969
                        -> showVal1 u <> text ".1"
      VFst u
                        -> showVal1 u <> text ".2"
970
      VSnd u
```

```
971
       VPathP v0 v1 v2 -> text "PathP" <+> showVals [v0,v1,v2]
 972
       VAppFormula v phi -> showVal v <+> char '@' <+> showFormula phi
 973
       VComp v0 v1 vs
                      ->
 974
        text "comp" <+> showVals [v0,v1] <+> text (C.showSystem vs)
 975
                        -> text "Glue" <+> showVal1 a <+> text (C.showSystem ts)
       VGlue a ts
                        -> text "glue" <+> showVal1 a <+> text (C.showSystem ts)
 976
       VGlueElem a ts
       VUnGlueElem a ts -> text "unglue" <+> showVal1 a <+> text (C.showSystem ts)
 977
       VUnGlueElemU v b es -> text "unglue U" <+> showVals [v,b]
 978
 979
                             <+> text (C.showSystem es)
 980
                        -> text "comp (< > U)" <+> showVal1 a <+> text (C.showSystem ts)
       VCompU a ts
                          -> text "Id" <+> showVals [a,u,v]
 981
      VId a u v
                          -> text "idC" <+> showVal1 b <+> text (C.showSystem ts)
       VIdPair b ts
 982
                        -> text "idJ" <+> showVals [a,t,c,d,x,p]
 983
       VIdJ a t c d x p
 984
 985 showPLam :: Val -> Doc
 986 showPLam e = case e of
 987
      VPLam i a@VPLam{} -> text (show i) <+> showPLam a
 988
      VPLam i a
                        -> text (show i) <> char '>' <+> showVal a
 989
                        -> showVal e
 990
 991 -- Merge lambdas of the same type
 992 showLam :: Val -> Doc
 993 showLam e = case e of
      VLam x t a@(VLam t'
 995
         | t == t'
                   -> text x <+> showLam a
 996
         | otherwise ->
 997
          text x <+> colon <+> showVal t <> char ')' <+> text "->" <+> showVal a
 998
       VPi _ (VLam x t a@(VPi _ (VLam _ t' _)))
999
        | t == t' -> text x <+> showLam a
1000
         | otherwise ->
1001
          text x <+> colon <+> showVal t <> char ')' <+> text "->" <+> showVal a
1002
      VLam x t e
                         ->
1003
        text x <+> colon <+> showVal t <> char ')' <+> text "->" <+> showVal e
1004
      VPi (VLam x t e) ->
1005
        text x <+> colon <+> showVal t <> char ')' <+> text "->" <+> showVal e
       _ -> showVal e
1006
1007
1008 showVal1 :: Val -> Doc
1009 showVall v = case v of
1010
                        -> showVal v
      VU
1011
                        -> showVal v
      VCon c []
1012
                        -> showVal v
      VVar{}
                        -> showVal v
1013
      VFst{}
1014
                        -> showVal v
      VSnd{}
1015
      Ter t rho | isEmpty (showEnv False rho) -> showTer1 t
1016
                        -> parens (showVal v)
1017
1018 showVals :: [Val] -> Doc
1019 showVals = hsep . map showVal1
1020 :::::::::::
1021 Eval.hs
1022 :::::::::::
1023 {-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}
1024 module Eval where
1025
1026 import Data.List
1027 import Data.Maybe (fromMaybe)
1028 import Data.Map (Map,(!),mapWithKey,assocs,filterWithKey
                    ,elems,intersectionWith,intersection,keys
1029
                     ,member,notMember,empty)
1030
1031 import qualified Data. Map as Map
1032 import qualified Data. Set as Set
1033
1034 import qualified Connections as C
1035 import CTT
1036
                           -----
1037 ------
1038 -- Lookup functions
1039
1040 look :: String -> Env -> Val
```

```
1041 look x (Env (Upd y rho, v: vs, fs, os)) | x == y = v
                                          | otherwise = look x (Env (rho, vs, fs, os))
1042
1043 look x r@(Env (Def \_ decls rho,vs,fs,C.Nameless os)) = case lookup x decls of
1044
       Just (_,t) -> eval r t
1045
       Nothing
                  -> look x (Env (rho, vs, fs, C. Nameless os))
1046 look x (Env (Sub \_ rho,vs,\_:fs,os)) = look x (Env (rho,vs,fs,os))
1047 look x (Env (Empty,_,_,)) = error  "look: not found " ++ show x
1048
1049 lookType :: String -> Env -> Val
1050 lookType x (Env (Upd y rho,v:vs,fs,os))
       | x /= y
                     = lookType x (Env (rho,vs,fs,os))
1052
       | VVar _ a <- v = a
       | otherwise = error ""
1053
1054 lookType x r@(Env (Def _ decls rho,vs,fs,os)) = case lookup x decls of 1055    Just (a,_) -> eval r a
       Nothing -> lookType x (Env (rho,vs,fs,os))
1056
1057 lookType x (Env (Sub _ rho,vs,_:fs,os)) = lookType x (Env (rho,vs,fs,os))
1058 lookType x (Env (Empty,_,_,_))
                                        = error $ "lookType: not found " ++ show x
1060 lookName :: C.Name -> Env -> C.Formula
1061 lookName i (Env (Upd \_ rho,v:vs,fs,os)) = lookName i (Env (rho,vs,fs,os)) 1062 lookName i (Env (Def \_ rho,vs,fs,os)) = lookName i (Env (rho,vs,fs,os))
1063 lookName i (Env (Sub \bar{j} rho,vs,phi:fs,os)) | i == j
                                                           = phi
                                                | otherwise = lookName i (Env (rho, vs, fs, os))
1065 lookName i _ = error $ "lookName: not found " ++ show i
1066
1067
1068
       ______
1069 -- Nominal instances
1070
1071 instance C.Nominal Ctxt where
1072
     support _ = []
       act e _ = e
swap e _ = e
1073
1074
1075
1076 instance C.Nominal Env where
1077
       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
1078
       swap (Env (rho, vs, fs, os)) ij = Env $ C.swap (rho, vs, fs, os) ij
1079
1080
1081 instance C. Nominal Val where
       support v = case \ v \ of
1082
         VU
1083
                                  -> []
         Ter _ e
                                  -> C.support e
1084
         VPi u v
1085
                                 -> C.support [u,v]
         VComp a u ts
1086
                                 -> C.support (a,u,ts)
1087
         VPathP a v0 v1
                                -> C.support [a,v0,v1]
1088
                                 -> i `delete` C.support v
         VPLam i v
1089
         VSigma u v
                                 -> C.support (u,v)
1090
         VPair u v
                                 -> C.support (u,v)
                                 -> C.support u
1091
         VFst u
1092
         VSnd u
                                 -> C.support u
         VCon _ vs
1093
                                 -> C.support vs
1094
1095
         VVar _ v
1096
                                 -> C.support v
         V0paque _ v
1097
                                 -> C.support v
1098
         VApp u v
                                 -> C.support (u,v)
1099
         VLam _ u v
                                 -> C.support (u,v)
                            -> C.support (u,v)
-> C.support (u,phi)
1100
         VAppFormula u phi
1101
         VSplit u v
                                 -> C.support (u,v)
1102
         VGlue a ts
                                 -> C.support (a,ts)
                              -> C.support (a,ts)
-> C.support (a,ts)
1103
         VGlueElem a ts
1104
         VUnGlueElem a ts
1105
         VCompU a ts
                                 -> C.support (a,ts)
         VUnGlueElemU a b es -> C.support (u,ts)

VIdPair u us -> C.support (u,ts)
1106
         VIdPair u us
                                 -> C.support (u,us)
1107
         VId a u v
                                 -> C.support (a,u,v)
1108
         VIdJ a u c d x p -> C.support [a,u,c,d,x,p]
1109
1110
```

```
act u (i, phi) | i `notElem` C.support u = u
1111
1112
                      | otherwise =
1113
         let acti :: C.Nominal a => a -> a
1114
             acti u = C.act u (i, phi)
1115
             sphi = C.support phi
1116
         in case u of
1117
              VU
                           -> VU
                           -> Ter t (acti e)
1118
              Ter t e
                           -> VPi (acti a) (acti f)
1119
              VPi a f
1120
              VComp a v ts -> compLine (acti a) (acti v) (acti ts)
1121
              VPathP a u v -> VPathP (acti a) (acti u) (acti v)
1122
              VPLam j v | j == i -> u
                        j `notElem` sphi -> VPLam j (acti v)
1123
                        | otherwise -> VPLam k (acti (v `C.swap` (j,k)))
1124
1125
                   where k = C.fresh (v,C.Atom i,phi)
1126
                                      -> VSigma (acti a) (acti f)
              VSigma a f
1127
              VPair u v
                                      -> VPair (acti u) (acti v)
                                      -> fstVal (acti u)
1128
              VFst u
                                      -> sndVal (acti u)
1129
              VSnd u
1130
              VCon c vs
                                      -> VCon c (acti vs)
1131
              VPCon c a vs phis
                                      -> pcon c (acti a) (acti vs) (acti phis)
1132
              VHComp a u us
                                      -> hComp (acti a) (acti u) (acti us)
1133
              VVar x v
                                      -> VVar x (acti v)
1134
              V0paque x v
                                      -> VOpaque x (acti v)
1135
              VAppFormula u psi
                                      -> acti u @@ acti psi
1136
              VApp u v
                                      -> app (acti u) (acti v)
              VLam x t u
                                      -> VLam x (acti t) (acti u)
1137
                                      -> app (acti u) (acti v)
1138
              VSplit u v
1139
              VGlue a ts
                                      -> glue (acti a) (acti ts)
              VGlueElem a ts
1140
                                      -> glueElem (acti a) (acti ts)
1141
              VUnGlueElem a ts
                                      -> unglueElem (acti a) (acti ts)
1142
              VUnGlueElemU a b es
                                      -> unGlueU (acti a) (acti b) (acti es)
1143
              VCompU a ts
                                      -> compUniv (acti a) (acti ts)
1144
              VIdPair u us
                                      -> VIdPair (acti u) (acti us)
                                       -> VId (acti a) (acti u) (acti v)
1145
              VId a u v
1146
              VIdJ a u c d x p
1147
                idJ (acti a) (acti u) (acti c) (acti d) (acti x) (acti p)
1148
1149
       -- This increases efficiency as it won't trigger computation.
1150
       swap u ij@(i,j) =
         let sw :: C.Nominal a => a -> a
1151
1152
             sw u = C.swap u ij
1153
         in case u of
1154
              VU
                                      -> VU
                                      -> Ter t (sw e)
1155
              Ter t e
                                      -> VPi (sw a) (sw f)
1156
              VPi a f
              VComp a v ts
                                      -> VComp (sw a) (sw v) (sw ts)
1157
1158
              VPathP a u v
                                      -> VPathP (sw a) (sw u) (sw v)
                                      -> VPLam (C.swapName k ij) (sw v)
1159
              VPLam k v
1160
              VSigma a f
                                      -> VSigma (sw a) (sw f)
                                      -> VPair (sw u) (sw v)
1161
              VPair u v
                                      -> VFst (sw u)
1162
              VFst u
              VSnd u
                                      -> VSnd (sw u)
1163
              VCon c vs
                                      -> VCon c (sw vs)
1164
1165
              VPCon c a vs phis
                                      -> VPCon c (sw a) (sw vs) (sw phis)
                                      -> VHComp (sw a) (sw u) (sw us)
1166
              VHComp a u us
1167
              VVar x v
                                      -> VVar x (sw v)
1168
              V0paque x v
                                      -> VOpaque x (sw v)
1169
              VAppFormula u psi
                                      -> VAppFormula (sw u) (sw psi)
                                      -> VApp (sw u) (sw v)
1170
              VApp u v
1171
              VLam x u v
                                      -> VLam x (sw u) (sw v)
1172
              VSplit u v
                                      -> VSplit (sw u) (sw v)
1173
              VGlue a ts
                                      -> VGlue (sw a) (sw ts)
                                      -> VGlueElem (sw a) (sw ts)
1174
              VGlueElem a ts
1175
                                      -> VUnGlueElem (sw a) (sw ts)
              VUnGlueElem a ts
1176
              VUnGlueElemU a b es
                                      -> VUnGlueElemU (sw a) (sw b) (sw es)
1177
              VCompU a ts
                                      -> VCompU (sw a) (sw ts)
                                      -> VIdPair (sw u) (sw us)
1178
              VIdPair u us
              VId a u v
1179
                                      -> VId (sw a) (sw u) (sw v)
1180
              VIdJ a u c d x p
                                      ->
```

```
1181
               VIdJ (sw a) (sw u) (sw c) (sw d) (sw x) (sw p)
1182
1183 -----
1184 -- The evaluator
1185
1186 eval :: Env -> Ter -> Val
1187 eval rho@(Env (\_,\_,\_,C.Nameless os)) v = case v of
1188
      U
                          -> VU
1189
      App r s
                          -> app (eval rho r) (eval rho s)
1190
      Var i
        | i `Set.member` os -> VOpaque i (lookType i rho)
1191
        1192
1193
       Pi t@(Lam _ a _)
      Sigma t@(Lam _ a _) -> VSigma (eval rho a) (eval rho t)
1194
                        -> VPair (eval rho a) (eval rho b)
1195
       Pair a b
1196
                         -> fstVal (eval rho a)
       Fst a
                          -> sndVal (eval rho a)
1197
      Snd a
1198
      Where t decls
                          -> eval (defWhere decls rho) t
1199
      Con name ts
                          -> VCon name (map (eval rho) ts)
1200
      PCon name a ts phis ->
1201
        pcon name (eval rho a) (map (eval rho) ts) (map (evalFormula rho) phis)
1202
       Lam{}
                          -> Ter v rho
                          -> Ter v rho
1203
       Split{}
1204
       Sum{}
                          -> Ter v rho
                          -> Ter v rho
1205
      HSum{}
                          -> Ter v rho
1206
      Undef{}
                          -> Ter v rho
1207
      Hole{}
      PathP a e0 e1
1208
                          -> VPathP (eval rho a) (eval rho e0) (eval rho e1)
1209
      PLam i t
                          -> let j = C.fresh rho
1210
                             in VPLam j (eval (sub (i,C.Atom j) rho) t)
1211
      AppFormula e phi -> eval rho e @@ evalFormula rho phi
1212
      Comp a t0 ts
1213
        compLine (eval rho a) (eval rho t0) (evalSystem rho ts)
1214
      HComp a t0 ts
1215
        hComp (eval rho a) (eval rho t0) (evalSystem rho ts)
1216
      Fill a t0 ts
1217
        fillLine (eval rho a) (eval rho t0) (evalSystem rho ts)
1218
                          -> glue (eval rho a) (evalSystem rho ts)
       Glue a ts
1219
                          -> glueElem (eval rho a) (evalSystem rho ts)
      GlueElem a ts
1220
                          -> unglueElem (eval rho a) (evalSystem rho ts)
      UnGlueElem a ts
                          -> VId (eval rho a) (eval rho r) (eval rho s)
1221
      Idars
                          -> VIdPair (eval rho b) (evalSystem rho ts)
1222
      IdPair b ts
1223
      IdJatcdxp
                          -> idJ (eval rho a) (eval rho t) (eval rho c)
1224
                                  (eval rho d) (eval rho x) (eval rho p)
1225
                          -> error $ "Cannot evaluate " ++ show v
1226
1227 evals :: Env -> [(Ident, Ter)] -> [(Ident, Val)]
1228 evals env bts = [(b,eval env t) | (b,t) <- bts]
1229
1230 evalFormula :: Env -> C.Formula -> C.Formula
1231 evalFormula rho phi = case phi of
                    -> lookName i rho
1232
      C.Atom i
      C.NegAtom i
1233
                       -> C.negFormula (lookName i rho)
      phi1 C.:/\: phi2 -> evalFormula rho phi1 `C.andFormula` evalFormula rho phi2
1234
      phil C.:\/: phi2 -> evalFormula rho phil `C.orFormula` evalFormula rho phi2
1235
1236
                     -> phi
1237
1238 evalSystem :: Env -> C.System Ter -> C.System Val
1239 evalSystem rho ts =
      let out = concat [ let betas = C.meetss [ C.invFormula (lookName i rho) d
1240
                         | (i,d) <- assocs alpha ]
in [ (beta,eval (rho `C.face` beta) talpha) | beta <- betas ]</pre>
1241
1242
1243
                        | (alpha,talpha) <- assocs ts ]</pre>
      in C.mkSystem out
1244
1245
1246 app :: Val -> Val -> Val
1247 app u v = case(u,v) of
       (Ter (Lam x _ t) e,_)
1248
                                          -> 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
1249
1250
```

```
-> error $ "app: missing case in split for " ++ c
1251
       (Ter (Split _ _ _ nvs) e, VPCon c _ us phis) -> case lookupBranch c nvs of
1252
1253
         Just (PBranch _ xs is t) -> eval (subs (zip is phis) (upds (zip xs us) e)) t
           -> error $ "app: missing case in split for " ++ c
1254
                      _ ty hbr) e,VHComp a w ws) -> case eval e ty of
1255
       (Ter (Split _
         VPi _ f \rightarrow let j = C.fresh (e,v)
1256
1257
                        wsj = Map.map (@@ j) ws
1258
                        w'
                           = app u w
                         ws' = mapWithKey (\alpha -> app (u `C.face` alpha)) wsj
1259
1260
                         -- a should be constant
1261
                     in comp j (app f (fill j a w wsj)) w' ws'
1262
           -> error $ "app: Split annotation not a Pi type " ++ show u
1263
       (Ter Split{} _,_) | isNeutral v
                                               -> VSplit u v
1264
       (VComp (VPLam i (VPi a f)) li0 ts,vi1) ->
1265
                    = C.fresh (u,vi1)
             (aj,fj) = (a,f) `C.swap` (i,j)
1266
                     = Map.map (@@ j) ts
1267
             tsi
1268
                     = transFillNeg j aj vi1
             V
1269
             vi0
                     = transNeg j aj vil
1270
         in comp j (app fj v) (app li0 vi0)
1271
                   (intersectionWith app tsj (C.border v tsj))
       _ | isNeutral u
1272
                             -> VApp u v
                              -> error $ "app \n " ++ show u ++ "\n " ++ show v
1273
1274
1275 fstVal, sndVal :: Val -> Val
1276 fstVal (VPair a b)
                           = a
1277 fstVal u | isNeutral u = VFst u
                            = error $ "fstVal: " ++ show u ++ " is not neutral."
1278 fstVal u
1279 sndVal (VPair a b)
                            = b
1280 sndVal u | isNeutral u = VSnd u
                             = error $ "sndVal: " ++ show u ++ " is not neutral."
1281 sndVal u
1282
1283 -- infer the type of a neutral value
1284 inferType :: Val -> Val
1285 inferType v = case v of
1286
       VVar _ t -> t
       VOpaque _ t -> t
Ter (Undef _ t) rho -> eval rho t
1287
1288
1289
       VFst t -> case inferType t of
1290
         VSigma a _ -> a
                    -> error $ "inferType: expected Sigma type for " ++ show v
1291
                       ++ ", got " ++ show ty
1292
1293
       VSnd t -> case inferType t of
         VSigma _ f -> app f (VFst t)
1294
1295
                    -> error $ "inferType: expected Sigma type for " ++ show v
1296
                       ++ ", got " ++ show ty
       VSplit s@(Ter (Split _ _ t _) rho) v1 -> case eval rho t of
1297
1298
         VPi _ f -> app f v1
1299
                 -> error $ "inferType: Pi type expected for split annotation in "
1300
                    ++ show v ++ ", got " ++ show ty
1301
       VApp t0 t1 -> case inferType t0 of
1302
         VPi _ f -> app f t1
1303
                 -> error $ "inferType: expected Pi type for " ++ show v
                    ++ ", got " ++ show ty
1304
1305
       VAppFormula t phi -> case inferType t of
1306
         VPathP a _ _ -> a @@ phi
1307
                     -> error $ "inferType: expected PathP type for " ++ show v
1308
                       ++ ", got " ++ show ty
1309
       VComp a _
                   -> a @@ C.One
         VUnGlueElem _ b _ -> b
                                   -- This is wrong! Store the type??
1310 --
1311
       VUnGlueElemU _ b _ -> b
       VIdJ _ _ c _ x p -> app (app c x) p _ -> error $ "inferType: not neutral " ++ show v
1312
1313
1314
1315 (@@) :: C.ToFormula a => Val -> a -> Val
                               = u `C.act` (i,C.toFormula phi)
1316 (VPLam i u) @@ phi
                               = VAppFormula v (C.toFormula phi)
1317 v@(Ter Hole{} _) @@ phi
                               = case (inferType v,C.toFormula phi) of
1318 v @@ phi | isNeutral v
       (VPathP _ a0 _,C.Dir 0) -> a0 (VPathP _ a1,C.Dir 1) -> a1
1319
1320
```

```
1321
                            -> VAppFormula v (C.toFormula phi)
1322 v @@ phi
                                = error $ "(@@): " ++ show v ++ " should be neutral."
1323
1324 -- Applying a *C.fresh* name.
1325 (@@@) :: Val -> C.Name -> Val
1326 (VPLam i u) @@@ j = u C.swap (i,j)
                      = VAppFormula v (C.toFormula j)
1327 v @@@ j
1328
1329
1330 -----
1331 -- Composition and filling
1333 comp :: C.Name -> Val -> C.System Val -> Val
1334 comp i a u ts | C.eps `member` ts = (ts ! C.eps) `C.face` (i C.~> 1)
1335 comp i a u ts = case a of
       VPathP p v0 v1 -> let j = C.fresh (C.Atom i,a,u,ts)
1336
1337
                         in VPLam j $ comp i (p @@ j) (u @@ j) $
1338
                              C.insertsSystem [(j C. \sim 0, v0),(j C. \sim 1, v1)] (Map.map (@@ j)
ts)
1339
       VId b v0 v1 -> case u of
1340
         VIdPair r _ | all isIdPair (elems ts) ->
           let j = \overline{C}.fresh (C.Atom i,a,u,ts)
1341
1342
               VIdPair z = @@@ phi = z @@ phi
               sys (VIdPair _ ws) = ws
1343
               w = VPLam j $ comp i b (r @@ j) $
1344
1345
                               C.insertsSystem [(j \ C.\sim> 0, v0), (j \ C.\sim> 1, v1)]
1346
                                 (Map.map (@@@ j) ts)
           in VIdPair w (C.joinSystem (Map.map sys (ts `C.face` (i C.~> 1))))
1347
1348
           -> VComp (VPLam i a) u (Map.map (VPLam i) ts)
1349
       VSigma a f -> VPair uil comp_u2
         where (t1s, t2s) = (Map.map fstVal ts, Map.map sndVal ts)
1350
               (u1, u2) = (fstVal u, sndVal u)
1351
1352
                         = fill i a u1 t1s
               fill_u1
1353
                          = comp i a u1 t1s
               ui1
1354
               comp u2
                          = comp i (app f fill_u1) u2 t2s
1355
       VPi{} -> VComp (VPLam i a) u (Map.map (VPLam i) ts)
1356
       VU -> compUniv u (Map.map (VPLam i) ts)
1357
       VCompU a es | not (isNeutralU i es u ts) -> compU i a es u ts
1358
       VGlue b equivs | not (isNeutralGlue i equivs u ts) -> compGlue i b equivs u ts
1359
       Ter (Sum _ _ nass) env -> case u of
         VCon n us | all isCon (elems ts) -> case lookupLabel n nass of
1360
1361
           Just as -> let tsus = C.transposeSystemAndList (Map.map unCon ts) us
1362
                      in VCon n $ comps i as env tsus
1363
           Nothing -> error $ "comp: missing constructor in labelled sum " ++ n
1364
           -> VComp (VPLam i a) u (Map.map (VPLam i) ts)
       Ter (HSum _
1365
                    nass) env -> compHIT i a u ts
       _ -> VComp (VPLam i a) u (Map.map (VPLam i) ts)
1366
1367
1368 compNeg :: C.Name -> Val -> Val -> C.System Val -> Val
1369 compNeg i a u ts = comp i (a `C.sym` i) u (ts `C.sym` i)
1370
1371 compLine :: Val -> Val -> C.System Val -> Val
1372 compLine a u ts = comp i (a @@ i) u (Map.map (@@ i) ts)
1373
       where i = C.fresh(a,u,ts)
1374
1375 compConstLine :: Val -> Val -> C.System Val -> Val
1376 compConstLine a u ts = comp i a u (Map.map (@@ i) ts)
1377
       where i = C.fresh(a,u,ts)
1378
1379 comps :: C.Name -> [(Ident,Ter)] -> Env -> [(C.System Val,Val)] -> [Val]
1380 comps i [] _ []
                                    = []
1381 comps i ((x,a):as) e ((ts,u):tsus) =
1382
       let v = fill i (eval e a) u ts
1383
           vi1 = comp i (eval e a) u ts
1384
           vs = comps i as (upd (x,v) e) tsus
1385
       in vil : vs
1386 comps _ _ _ = error "comps: different lengths of types and values"
1387
1388 fill :: C.Name -> Val -> Val -> C.System Val -> Val
1389 \text{ fill i a u ts} =
```

```
comp j (a `C.conj` (i,j)) u (C.insertSystem (i C.~> 0) u (ts `C.conj` (i,j)))
1390
1391
      where j = C.fresh (C.Atom i,a,u,ts)
1392
1393 fillNeg :: C.Name -> Val -> Val -> C.System Val -> Val
1394 fillNeg i a u ts = (fill i (a `C.sym` i) u (ts `C.sym` i)) `C.sym` i
1395
1396 fillLine :: Val -> Val -> C.System Val -> Val
1397 fillLine a u ts = VPLam i $ fill i (a @@ i) u (Map.map (@@ i) ts)
      where i = C.fresh(a,u,ts)
1398
1399
1400 -- fills :: C.Name -> [(Ident,Ter)] -> Env -> [(C.System Val,Val)] -> [Val]
1401 -- fills i [] _ [] = [] 1402 -- fills i ((x,a):as) e ((ts,u):tsus) =
1403 -- let v = fill i (eval e a) ts u
1404 --
          vs = fills i as (Upd e (x,v)) tsus
1405 -- in v : vs
1406 -- fills _ _ _ = error "fills: different lengths of types and values"
1407
1408
1409 -----
1410 -- Transport and squeeze (defined using comp)
1412 trans :: C.Name -> Val -> Val -> Val
1413 trans i v0 v1 = comp i v0 v1 empty
1415 transNeg :: C.Name -> Val -> Val -> Val
1416 transNeg i a u = trans i (a `C.sym` i) u
1417
1418 transLine :: Val -> Val -> Val
1419 transLine u v = trans i (u @@ i) v
1420
      where i = C.fresh(u,v)
1421
1422 transNegLine :: Val -> Val -> Val
1423 transNegLine u v = transNeg i (u @@ i) v
1424
      where i = C.fresh(u,v)
1425
1426 -- TODO: define in terms of comps?
1427 transps :: C.Name -> [(Ident,Ter)] -> Env -> [Val] -> [Val]
1428 transps i [] _ [] = [] 1429 transps i ((x,a):as) e (u:us) =
     let v = transFill i (eval e a) u
1430
1431
          vil = trans i (eval e a) u
1432
          vs = transps i as (upd (x,v) e) us
1433
      in vi1 : vs
1434 transps _ _ _ = error "transps: different lengths of types and values"
1436 transFill :: C.Name -> Val -> Val -> Val
1437 transFill i a u = fill i a u empty
1438
1439 transFillNeg :: C.Name -> Val -> Val -> Val
1440 transFillNeg i a u = (transFill i (a `C.sym` i) u) `C.sym` i
1441
1442 -- Given u of type a "squeeze i a u" connects in the direction i
1443 -- trans i a u(i=0) to u(i=1)
1444 squeeze :: C.Name -> Val -> Val -> Val
1445 squeeze i a u = comp j (a C.disj (i,j)) u $ C.mkSystem [ (i C. > 1, uil) ]
1446
      where j = C.fresh (C.Atom i,a,u)
1447
            uil = u `C.face` (i C.~> 1)
1448
1449 squeezes :: C.Name -> [(Ident,Ter)] -> Env -> [Val] -> [Val]
1450 squeezes i xas e us = comps j xas (e `C.disj` (i,j)) us
1451
      where j = C.fresh (us,e,C.Atom i)
            us' = [(C.mkSystem [(i C.~> 1, u `C.face` (i C.~> 1))],u) | u <- us]
1452
1453
1454
1455 -----
1456 -- | Id
1458 idJ :: Val -> Val -> Val -> Val -> Val -> Val
1459 idJ a v c d x p = case p of
```

```
1460
       VIdPair w ws -> comp i (app (app c (w @@ i)) w') d
1461
                         (C.border d (C.shape ws))
1462
         where w' = VIdPair (VPLam j $ w @@ (C.Atom i C.:/\: C.Atom j))
1463
                       (C.insertSystem (i C.~> 0) v ws)
1464
               i:j:_ = C.freshs [a,v,c,d,x,p]
       \_ -> VIdJ a \overline{v} c d x p
1465
1466
1467 isIdPair :: Val -> Bool
1468 isIdPair VIdPair{} = True
1469 isIdPair _
                        = False
1470
1471
1472 -----
1473 -- | HITs
1474
1475 pcon :: LIdent -> Val -> [Val] -> [C.Formula] -> Val
1476 pcon c a@(Ter (HSum _ _ lbls) rho) us phis = case lookupPLabel c lbls of 1477 Just (tele,is,ts) | C.eps `member` vs -> vs ! C.eps
1478
                         | otherwise
                                           -> VPCon c a us phis
         where rho' = subs (zip is phis) (updsTele tele us rho)
1479
1480
                  = evalSystem rho' ts
              ٧S
                        -> error "pcon"
1481
       Nothing
1482 pcon c a us phi
                         = VPCon c a us phi
1484 compHIT :: C.Name -> Val -> Val -> C.System Val -> Val
1485 compHIT i a u us
1486
       | isNeutral u || isNeutralSystem us =
1487
           VComp (VPLam i a) u (Map.map (VPLam i) us)
1488
       | otherwise =
1489
           hComp (a `C.face` (i C.~> 1)) (transpHIT i a u) $
1490
             mapWithKey (\alpha uAlpha ->
1491
                          VPLam i $ squeezeHIT i (a `C.face` alpha) uAlpha) us
1492
1493 -- Given u of type a(i=0), transpHIT i a u is an element of a(i=1).
1494 transpHIT :: C.Name -> Val -> Val -> Val
1495 transpHIT i a@(Ter (HSum \_ \_ nass) env) u =
1496
     let j = C.fresh(a,u)
1497
          aij = C.swap a (i,j)
1498
1499
      case u of
1500
       VCon n us -> case lookupLabel n nass of
1501
         Just as -> VCon n (transps i as env us)
1502
         Nothing -> error $ "transpHIT: missing constructor in labelled sum " ++ n
       VPCon c _ ws0 phis -> case lookupLabel c nass of
1503
         Just as -> pcon c (a `C.face` (i C.\sim>1)) (transps i as env ws0) phis
1504
         Nothing -> error $ "transpHIT: missing path constructor " ++ c
1505
1506
       VHComp _ v vs ->
1507
         hComp (a `C.face` (i C.~> 1)) (transpHIT i a v) $
1508
           mapWithKey (\alpha vAlpha ->
1509
                        VPLam j $ transpHIT j (aij `C.face` alpha) (vAlpha @@ j)) vs
       _ -> error $ "transpHIT: neutral " ++ show u
1510
1511
1512 -- given u(i) of type a(i) "squeezeHIT i a u" connects in the direction i
1513 -- transHIT i a u(i=0) to u(i=1) in a(1)
1514 squeezeHIT :: C.Name -> Val -> Val -> Val
1515 squeezeHIT i a@(Ter (HSum _ _ nass) env) u =
1516
     let j = C.fresh(a,u)
1517
      in
1518
      case u of
1519
       VCon n us -> case lookupLabel n nass of
1520
         Just as -> VCon n (squeezes i as env us)
1521
         Nothing -> error $ "squeezeHIT: missing constructor in labelled sum " ++ n
1522
       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
1523
         Nothing -> error $ "squeezeHIT: missing path constructor " ++ c
1524
       VHComp _ v vs -> hComp (a `C.face` (i C.~> 1)) (squeezeHIT i a v) $
1525
           mapWithKey
1526
1527
             (\alpha vAlpha -> case Map.lookup i alpha of
               Nothing -> VPLam j $ squeezeHIT i (a `C.face` alpha) (vAlpha @@ j)
1528
1529
               Just C.Zero -> VPLam j $ transpHIT i
```

```
1530
                              (a `C.face` (Map.delete i alpha)) (vAlpha @@ j)
               Just C.One -> vAlpha)
1531
1532
             ٧S
       _ -> error $ "squeezeHIT: neutral " ++ show u
1533
1534
1535 hComp :: Val -> Val -> C.System Val -> Val
1536 hComp a u us | C.eps `member` us = (us ! C.eps) @@ C.One
                  | otherwise
1537
                                    = VHComp a u us
1538
1539 ----
1540 -- | Glue
1541
1542 -- An equivalence for a type a is a triple (t,f,p) where
1543 -- t : U
1544 -- f : t -> a
1545 -- p : (x : a) \rightarrow isContr((y:t) * Id a x (f y))
1546 -- with isContr c = (z : c) * ((z' : C) -> Id c z z')
1547
1548 -- Extraction functions for getting a, f, s and t:
1549 equivDom :: Val -> Val
1550 equivDom = fstVal
1551
1552 equivFun :: Val -> Val
1553 equivFun = fstVal . sndVal
1555 equivContr :: Val -> Val
1556 equivContr = sndVal . sndVal
1557
1558 glue :: Val -> C.System Val -> Val
1559 glue b ts | C.eps `member` ts = equivDom (ts ! C.eps)
               | otherwise
                                 = VGlue b ts
1560
1561
1562 glueElem :: Val -> C.System Val -> Val
1563 glueElem v us | C.eps `member` us = us ! C.eps
1564 glueElem v us = VGlueElem v us
1565
1566 unglueElem :: Val -> C.System Val -> Val
1567 unglueElem w isos | C.eps `member` isos = app (equivFun (isos ! C.eps)) w
1568
                       | otherwise
                                           = case w of
1569
                                                VGlueElem v us -> v
1570
                                                -> VUnGlueElem w isos
1571
1572 unGlue :: Val -> Val -> C.System Val -> Val
1573 unGlue w b equivs | C.eps `member` equivs = app (equivFun (equivs ! C.eps)) w
                       | otherwise
1574
                                             = case w of
1575
                                                  VGlueElem v us -> v
                                                  _ -> error ("unglue: neutral" ++ show w)
1576
1577
1578 isNeutralGlue :: C.Name -> C.System Val -> Val -> C.System Val -> Bool
1579 isNeutralGlue i equivs u0 ts = (C.eps `notMember` equivsi0 && isNeutral u0) ||
1580
       any (\(alpha,talpha) ->
                C.eps `notMember` (equivs `C.face` alpha) && isNeutral talpha)
1581
1582
         (assocs ts)
1583
       where equivsi0 = equivs `C.face` (i C.~> 0)
1584
1585 -- this is exactly the same as isNeutralGlue?
1586 isNeutralU :: C.Name -> C.System Val -> Val -> C.System Val -> Bool
1587 isNeutralU i eqs u0 ts = (C.eps `notMember` eqsi0 && isNeutral u0) ||
1588
       any (\(alpha,talpha) ->
1589
                C.eps `notMember` (eqs `C.face` alpha) && isNeutral talpha)
1590
         (assocs ts)
1591
       where eqsi0 = eqs `C.face` (i C.~> 0)
1592
1593 -- Extend the system ts to a total element in b given q : isContr b
1594 extend :: Val -> Val -> C.System Val -> Val
1595 extend b q ts = comp i b (fstVal q) ts'
1596
       where i = C.fresh(b,q,ts)
             ts' = mapWithKey
1597
1598
                     (\alpha tAlpha -> app ((sndVal q) `C.face` alpha) tAlpha @@ i) ts
1599
```

```
1600 -- psi/b corresponds to ws
1601 -- b0
            corresponds to wi0
1602 -- a0
             corresponds to vi0
1603 -- psi/a corresponds to vs
1604 -- a1'
            corresponds to vil'
1605 -- equivs' corresponds to delta
1606 -- til' corresponds to usil'
1607 compGlue :: C.Name -> Val -> C.System Val -> Val -> C.System Val -> Val
1608 compGlue i a equivs wi0 ws = glueElem vi1 usi1
      where ail = a `C.face` (i C.~> 1)
1609
1610
            vs = mapWithKey
1611
                    (\alpha wAlpha ->
                      unGlue wAlpha (a `C.face` alpha) (equivs `C.face` alpha)) ws
1612
1613
            vsil = vs `C.face` (i C.~> 1) -- same as: C.border vil vs
1614
            vi0 = unGlue wi0 (a `C.face` (i C.~> 0)) (equivs `C.face` (i C.~> 0)) -- in
1615
a(i0)
1616
1617
            vil' = comp i a vi0 vs
                                             -- in a(i1)
1618
1619
            equivsI1 = equivs `C.face` (i C.~> 1)
            equivs' = filterWithKey (\alpha _ -> i `notMember` alpha) equivs
1620
1621
1622
                   = mapWithKey (\gamma equivG ->
                       fill i (equivDom equivG) (wi0 `C.face` gamma) (ws `C.face` gamma))
1623
1624
                     equivs'
1625
            usil'
                  = mapWithKey (\gamma equivG ->
                       comp i (equivDom equivG) (wi0 `C.face` gamma) (ws `C.face` gamma))
1626
1627
                     equivs'
1628
1629
            -- path in ail between vil and f(il) usil' on equivs'
1630
                   = mapWithKey (\gamma equivG ->
1631
                       pathComp i (a `C.face` gamma) (vi0 `C.face` gamma)
                         (equivFun equivG `app` (us' ! gamma)) (vs `C.face` gamma))
1632
1633
1634
1635
            fibersys = intersectionWith VPair usil' ls' -- on equivs'
1636
            wsi1 = ws `C.face` (i C.~> 1)
1637
            fibersys' = mapWithKey
1638
              (\gamma equivG ->
1639
1640
                let fibsgamma = intersectionWith (\ x y -> VPair x (constPath y))
1641
                                  (wsil `C.face` gamma) (vsil `C.face` gamma)
                1642
1643
                     (fibsgamma `C.unionSystem` (fibersys `C.face` gamma))) equivsI1
1644
1645
1646
            vi1 = compConstLine ail vi1'
1647
                    (Map.map sndVal fibersys' `C.unionSystem` Map.map constPath vsil)
1648
1649
            usi1 = Map.map fstVal fibersys'
1650
1651 mkFiberType :: Val -> Val -> Val -> Val
1652 mkFiberType a x equiv = eval rho $
      Sigma $ Lam "y" tt (PathP (PLam (C.Name "_") ta) tx (App tf ty))
1653
      1654
1655
1656
                       ,("t",equivDom equiv)] emptyEnv
1657
1658 -- Assumes u' : A is a solution of us + (i0 -> u0)
1659 -- The output is an L-path in A(i1) between comp i u0 us and u'(i1)
1660 pathComp :: C.Name -> Val -> Val -> C.System Val -> Val
1661 pathComp i a u0 u' us = VPLam j $ comp i a u0 us'
      where j = C.fresh (C.Atom i,a,us,u0,u')
1662
            us' = C.insertsSystem [(j C.\sim>1, u')] us
1663
1664
1665 -
1666 -- | Composition in the Universe
1667
1668 -- any path between types define an equivalence
```

```
1669 eqFun :: Val -> Val -> Val
1670 egFun = transNegLine
1671
1672 unGlueU :: Val -> Val -> C.System Val -> Val
1673 unGlueU w b es | C.eps `Map.member` es = eqFun (es ! C.eps) w
                    | otherwise
1674
                                           = case w of
                                              VGlueElem v us
1675
1676
                                              -> VUnGlueElemU w b es
1677
1678 compUniv :: Val -> C.System Val -> Val
1679 compUniv b es | C.eps `Map.member` es = (es ! C.eps) @@ C.One
1680
                   I otherwise
                                         = VCompU b es
1681
1682 compU :: C.Name -> Val -> C.System Val -> Val -> C.System Val -> Val
1683 compU i a eqs wi0 ws = glueElem vi1 usi1
       where ail = a `C.face` (i C.~> 1)
1684
1685
             vs = mapWithKey
1686
                     (\alpha wAlpha ->
1687
                       unGlueU wAlpha (a `C.face` alpha) (eqs `C.face` alpha)) ws
1688
1689
             vsi1 = vs `C.face` (i C.~> 1) -- same as: C.border vil vs
             vi0 = unGlueU wi0 (a `C.face` (i C.~> 0)) (eqs `C.face` (i C.~> 0)) -- in a(i0)
1690
1691
1692
             vil' = comp i a vi0 vs
                                                -- in a(i1)
1693
1694
             eqsI1 = eqs `C.face` (i C.~> 1)
1695
             eqs' = filterWithKey (\alpha _ -> i `notMember` alpha) eqs
1696
1697
             IIS '
                    = mapWithKey (\gamma eqG ->
1698
                        fill i (eqG @@ C.One) (wi0 `C.face` gamma) (ws `C.face` gamma))
1699
                      eas '
1700
             usi1'
                   = mapWithKey (\gamma eqG ->
1701
                        comp i (eqG @@ C.One) (wi0 `C.face` gamma) (ws `C.face` gamma))
1702
                      eas '
1703
1704
             -- path in ail between vil and f(il) usil' on egs'
1705
                    = mapWithKey (\gamma eqG ->
                        pathComp i (a `C.face` gamma) (vi0 `C.face` gamma)
1706
1707
                           (eqFun eqG (us' ! gamma)) (vs `C.face` gamma))
1708
1709
1710
             fibersys = intersectionWith (\ x y -> (x,y)) usil' ls' -- on eqs'
1711
1712
             wsi1 = ws `C.face` (i C.~> 1)
             fibersys' = mapWithKey
1713
               (\gamma eqG ->
1714
                 let fibsgamma = intersectionWith (\ x \ y \ -> (x, constPath \ y))
1715
1716
                                    (wsi1 `C.face` gamma) (vsi1 `C.face` gamma)
                 in lemEq eqG (vil' `C.face` gamma)
1717
1718
                           (fibsgamma `C.unionSystem` (fibersys `C.face` gamma))) eqsI1
1719
1720
             vi1 = compConstLine ai1 vi1'
                      (Map.map snd fibersys' `C.unionSystem` Map.map constPath vsil)
1721
1722
             usi1 = Map.map fst fibersys'
1723
1724
1725 lemEq :: Val -> Val -> C.System (Val, Val) -> (Val, Val)
1726 lemEq eq b aps = (a,VPLam i (compNeg j (eq @@ j) p1 thetas'))
1727
      where
        i:j:_ = C.freshs (eq,b,aps)
1728
1729
        ta = eq @@ C.One
        pls = mapWithKey (\alpha (aa,pa) ->
1730
                   let eqaj = (eq `C.face` alpha) @@ j
1731
                       ba = b `C.face` alpha
1732
1733
                   in comp j eqaj (pa @@ i)
                         (C.mkSystem [ (i C.~>0, transFill j eqaj ba)
1734
1735
                                   , (i C.~>1,transFillNeg j eqaj aa)])) aps
1736
        thetas = mapWithKey (\alpha (aa,pa) ->
1737
                   let eqaj = (eq `C.face` alpha) @@ j
                       ba = b `C.face` alpha
1738
```

```
1739
                   in fill j eqaj (pa @@ i)
1740
                         (C.mkSystem [ (i C.~>0,transFill j eqaj ba)
1741
                                   , (i C.~>1,transFillNeg j eqaj aa)])) aps
1742
1743
        a = comp i ta (trans i (eq @@ i) b) pls
1744
        p1 = fill i ta (trans i (eq @@ i) b) p1s
1745
        thetas' = C.insertsSystem [ (i C.\sim>0,transFill j (eq @@ j) b)
1746
                                 , (i C.\sim>1, transFillNeg j (eq @@ j) a)] thetas
1747
1748
1749 -- Old version:
1750 -- This version triggers the following error when checking the normal form of corrUniv:
1751 -- Parsed "examples/nunivalence2.ctt" successfully!
1752 -- Resolver failed: Cannot resolve name !3 at position (7,30062) in module nunivalence2
1753 -- compU :: Name -> Val -> C.System Val -> Val -> C.System Val -> Val
1754 -- compU i b es wi0 ws = glueElem vi1'' usi1''
          where bi1 = b `C.face` (i C.~> 1)
1755 --
1756 --
                VS
                     = mapWithKey (\alpha wAlpha ->
                         unGlueU wAlpha (b `C.face` alpha) (es `C.face` alpha)) ws
1757 --
1758 --
                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
1759 --
b(i0)
1760
1761 --
                     = fill i b vi0 vs
                                                  -- in b
                                                  -- is v `C.face` (i C.~> 1) in b(i1)
1762 --
                vi1 = comp i b vi0 vs
1763
1764 --
                esI1 = es `C.face` (i C.~> 1)
                es' = filterWithKey (\alpha _ -> i `Map.notMember` alpha) es
1765 --
1766 --
                es'' = filterWithKey (\alpha _ -> alpha `Map.notMember` es) esI1
1767
1768 --
                us'
                       = mapWithKey (\gamma eGamma ->
                           fill i (eGamma @@ C.One) (wi0 `C.face` gamma) (ws `C.face` gamma))
1769 --
1770 --
                         es'
1771 --
                usil'
                       = mapWithKey (\gamma eGamma ->
                           comp i (eGamma @@ C.One) (wi0 `C.face` gamma) (ws `C.face` gamma))
1772 --
1773 --
                         es'
1774
                ls'
1775 --
                       = mapWithKey (\gamma eGamma ->
                             pathComp i (b `C.face` gamma) (v `C.face` gamma)
  (transNegLine eGamma (us' ! gamma)) (vs `C.face` gamma))
1776 --
1777 --
1778 --
1779
1780 --
                vi1' = compLine (constPath bi1) vi1
1781 --
                         (ls' `C.unionSystem` Map.map constPath vsi1)
1782
1783 --
                wsi1 = ws `C.face` (i C.~> 1)
1784
1785 --
                -- for gamma in es'', (i1) gamma is in es, so wsil gamma
1786 --
                -- is in the domain of isoGamma
1787 --
                uls'' = mapWithKey (\gamma eGamma ->
                          isoToEquivU (bil `C.face` gamma) eGamma
  ((usil' `C.face` gamma) `C.unionSystem` (wsil `C.face` gamma))
1788 --
1789 --
1790 --
                             (vi1' `C.face` gamma))
                          es''
1791 --
1792
1793 --
                vsi1' = Map.map constPath $ C.border vi1' es' `C.unionSystem` vsi1
1794
                vi1'' = compLine (constPath bil) vi1'
1795 --
1796 --
                           (Map.map snd uls'' `C.unionSystem` vsi1')
1797
                1798 --
1799 --
1800 --
                              else fst (uls'' ! gamma))
1801 --
                          esI1
1802
1803 -- IsoToEquiv, takes a line eq in U, a system us and a value v, s.t. f us =
1804 -- C.border v. Outputs (u,p) s.t. C.border u = us and a path p between v
1805 -- and f u, where f is transNegLine eq
1806 -- isoToEquivU :: Val -> Val -> C.System Val -> Val -> (Val, Val)
1807 -- isoToEquivU b eq us v = (u, VPLam \ i \ theta)
```

```
1808 --
         where i:j:_
                       = C.freshs (b,eq,us,v)
1809 --
               еi
                       = eq @@ j
1810 --
                       = eq @@ C.One
               а
1811 --
                       = mapWithKey (\alpha uAlpha ->
               WS
1812 --
                                          transFillNeg j (ej `C.face` alpha) uAlpha) us
1813 --
               П
                       = comp j ej v ws
1814 --
               W
                       = fill j ej v ws
1815 --
                       = C.insertSystem (i C.~> 0) w $
               XS
1816 --
                         C.insertSystem (i C.~> 1) (transFillNeg j ej u) $ ws
1817 --
               theta
                       = compNeg j ej u xs
1818
1819 -- Old version:
1820 -- isoToEquivU :: Val -> Val -> System Val -> Val -> (Val, Val)
1821 -- isoToEquivU b eq us v = (u, VPLam \ i \ theta'')
1822 --
                      = C.freshs (b,eq,us,v)
         where i:j:_
1823 --
                       = eq @@ C.One
               а
1824 --
                       = transLine
               g
1825 --
               f
                       = transNegLine
1826 --
                       = VPLam j $ compNeg i (e @@ i) (trans i (e @@ i) y)
               s e y
                           (C.mkSystem [(j C.~> 0, transFill j (e @@ j) y)
1827 --
1828 --
                                     ,(j C.~> 1, transFillNeg j (e @@ j)
1829 --
                                                 (trans j (e @@ j) y))])
1830 --
               t e x
                       1831 --
                           (C.mkSystem [(j C.~> 0, transFill j (e @@ j)
1832 --
                                                 (transNeg j (e @@ j) x))
1833 --
                                     (j C.\sim 1, transFillNeg j (e @@ j) x)])
1834 --
               gν
                       = q eq v
1835 --
               us'
                       = mapWithKey (\alpha uAlpha ->
1836 --
                                          t (eq `C.face` alpha) uAlpha @@ i) us
1837 --
               theta
                       = fill i a gv us'
1838 --
                       = comp i a gv us'
                                          -- Same as "theta `C.face` (i C.~> 1)"
               11
1839 --
               WS
                       = C.insertSystem (i C.~> 0) gv $
1840 --
                         C.insertSystem (i C.~> 1) (t eq u @@ j) $
1841 --
                         mapWithKey
1842 --
                           (\alpha uAlpha ->
1843 --
                             t (eq `C.face` alpha) uAlpha @@ (C.Atom i :/\: C.Atom j)) us
1844 --
               theta'
                       = compNeg j a theta ws
                       = C.insertSystem (i C.~> 0) (s eq v @@ j) $
1845 --
               XS
                         C.insertSystem (i C.~> 1) (s eq (f eq u) @@ j) $
1846 --
1847 --
                         mapWithKey
1848 --
                           (\alpha uAlpha ->
                              s (eq `C.face` alpha) (f (eq `C.face` alpha) uAlpha) @@ j) us
1849 --
               theta'' = comp j b (f eq theta') xs
1850 --
1851
1852
1853 -----
1854 -- | Conversion
1856 class Convertible a where
1857
      conv :: [String] -> a -> a -> Bool
1858
1859 isCompSystem :: (C.Nominal a, Convertible a) => [String] -> C.System a -> Bool
1860 isCompSystem ns ts = and [ conv ns (getFace alpha beta) (getFace beta alpha)
1861
                             | (alpha,beta) <- C.allCompatible (keys ts) ]</pre>
1862
        where getFace a b = C.face (ts ! a) (b `C.minus` a)
1863
1864 instance Convertible Env where
1865
      conv ns (Env (rho1,vs1,fs1,os1)) (Env (rho2,vs2,fs2,os2)) =
1866
          conv ns (rho1, vs1, fs1, os1) (rho2, vs2, fs2, os2)
1867
1868 instance Convertible Val where
1869
      conv ns u v | u == v
1870
                   | otherwise =
1871
        let j = C.fresh(u,v)
1872
        in case (u,v) of
1873
           (Ter (Lam x a u) e,Ter (Lam x' a' u') e') ->
            let v@(VVar n _) = mkVarNice ns x (eval e a)
1874
             in conv (n:ns) (eval (upd (x,v) e) u) (eval (upd (x',v) e') u')
1875
1876
           (Ter (Lam x a u) e,u') ->
1877
            let v@(VVar n _) = mkVarNice ns x (eval e a)
```

```
1878
               in conv (n:ns) (eval (upd (x,v) e) u) (app u' v)
1879
             (u',Ter(Lam x a u) e) ->
1880
               let v@(VVar n _) = mkVarNice ns x (eval e a)
               in conv (n:ns) (app u' v) (eval (upd (x,v) e) u)
1881
             (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'
1882
1883
1884
1885
1886
1887
             -- (Ter Hole{} e,_) -> True
             -- (_,Ter Hole{} e') -> True
1888
             (VPi u v, VPi u' v') ->
1889
               let w@(VVar n _) = mkVarNice ns "X" u
in conv ns u u' && conv (n:ns) (app v w) (app v' w)
1890
1891
             (VSigma u v, VSigma u' v') ->
1892
               let w@(VVar n _) = mkVarNice ns "X" u
in conv ns u u' && conv (n:ns) (app v w) (app v' w)
1893
1894
             (VCon c us, VCon c' us') -> (c == c') && conv ns us us'
1895
             (VPCon c v us phis, VPCon c' v' us' phis') ->
1896
                (c == c') \&\& conv ns (v,us,phis) (v',us',phis')
1897
                                             -> conv ns u u' && conv ns v v'
             (VPair u v, VPair u' v')
1898
1899
                                              -> conv ns u (fstVal w) && conv ns v (sndVal w)
             (VPair u v,w)
1900
             (w, VPair u v)
                                              -> conv ns (fstVal w) u && conv ns (sndVal w) v
1901
             (VFst u, VFst u')
                                              -> conv ns u u'
1902
             (VSnd u, VSnd u')
                                             -> conv ns u u'
             (VApp u v, VApp u' v')
                                            -> conv ns u u' && conv ns v v'
1903
             (VSplit u v, VSplit u' v') → conv ns u u' && conv ns v v'
1904
             (V0paque x _, V0paque x' _) -> x == x'
1905
             (VVar x _, VVar x' _)
1906
                                              -> x == x'
1907
             (VPathP a b c, VPathP a' b' c') -> conv ns a a' && conv ns b b' && conv ns c c'
                                           -> conv ns (a `C.swap` (i,j)) (a' `C.swap` (i',j))
-> conv ns (a `C.swap` (i,j)) (p' @@ j)
1908
             (VPLam i a, VPLam i' a')
1909
             (VPLam i a,p')
1910
             (p,VPLam i' a')
                                              -> conv ns (p @@ j) (a' `C.swap`
                                                                                      (i',j))
1911
             (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')
1912
1913
1914
             (VGlueElem (VUnGlueElem b equivs) ts,g) -> conv ns (C.border b equivs,b) (ts,g)
1915
             (g, VGlueElem (VUnGlueElem b equivs) ts) -> conv ns (C.border b equivs,b) (ts,g)
1916
             (VGlueElem (VUnGlueElemU b _ equivs) ts,g) -> conv ns (C.border b equivs,b) (ts,g)
1917
             (g,VGlueElem (VUnGlueElemU b _ equivs) ts) -> conv ns (C.border b equivs,b) (ts,g)
1918
             (VGlueElem u us, VGlueElem u' us') -> conv ns (u,us) (u',us') (VUnGlueElemU u _ _, VUnGlueElemU u' _ _) -> conv ns u u'
1919
1920
             (VUnGlueElem u _,VUnGlueElem u' _) -> conv ns u u'
1921
1922
             (VCompU u es, VCompU u' es')
                                                              -> conv ns (u,es) (u',es')
             (VIdPair v vs, VIdPair v' vs')
1923
                                                            -> conv ns (v,vs) (v',vs')
1924
             (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') ->
1925
1926
                conv ns [a,u,c,d,x,p] [a',u',c',d',x',p']
1927
                                                             -> False
1928
1929 instance Convertible Ctxt where
      conv _ _ _ = True
1930
1931
1932 instance Convertible () where
1933
       conv _ _ _ = True
1934
1935 instance (Convertible a, Convertible b) => Convertible (a, b) where
1936
        conv ns (u, v) (u', v') = conv ns u u' & conv ns v v'
1937
1938 instance (Convertible a, Convertible b, Convertible c)
             => Convertible (a, b, c) where
1939
1940
        conv ns (u, v, w) (u', v', w') = conv ns <math>(u, (v, w)) (u', (v', w'))
1941
1942 instance (Convertible a, Convertible b, Convertible c, Convertible d)
1943
             => Convertible (a,b,c,d) where
        conv ns (u,v,w,x) (u',v',w',x') = conv ns <math>(u,v,(w,x)) (u',v',(w',x'))
1944
1945
1946 instance Convertible a => Convertible [a] where
        conv ns us us' = length us == length us' &&
1947
```

```
and [conv ns u u' | (u,u') \leftarrow zip us us']
1948
1949
1950 instance Convertible a => Convertible (C.System a) where
      conv ns ts ts' = keys ts == keys ts' &&
1952
                       and (elems (intersectionWith (conv ns) ts ts'))
1953
1954 instance Convertible C.Formula where
1955
      conv _ phi psi = C.dnf phi == C.dnf psi
1956
1957 instance Convertible (C.Nameless a) where
1958
      conv _ _ = True
1959
1960 -----
1961 -- | Normalization
1962
1963 class Normal a where
1964
      normal :: [String] -> a -> a
1965
1966 instance Normal Env where
1967
      normal ns (Env (rho, vs, fs, os)) = Env (normal ns (rho, vs, fs, os))
1968
1969 instance Normal Val where
1970
      normal ns v = case \ v \ of
1971
        VU
                             -> VU
1972
         Ter (Lam x t u) e
                             ->
1973
          let w = eval e t
1974
              v@(VVar n _) = mkVarNice ns x w
1975
          in VLam n (normal ns w) \$ normal (n:ns) (eval (upd (x,v) e) u)
1976
        Ter t e
                            -> Ter t (normal ns e)
1977
        VPi u v
                            -> VPi (normal ns u) (normal ns v)
                            -> VSigma (normal ns u) (normal ns v)
1978
        VSigma u v
1979
        VPair u v
                            -> VPair (normal ns u) (normal ns v)
1980
        VCon n us
                            -> VCon n (normal ns us)
1981
         VPCon n u us phis
                            -> VPCon n (normal ns u) (normal ns us) phis
1982
         VPathP a u0 u1
                            -> VPathP (normal ns a) (normal ns u0) (normal ns u1)
1983
        VPLam i u
                            -> VPLam i (normal ns u)
                            -> VComp (normal ns u) (normal ns v) (normal ns vs)
1984
        VComp u v vs
                            -> VHComp (normal ns u) (normal ns v) (normal ns vs)
1985
        VHComp u v vs
                            -> VGlue (normal ns u) (normal ns equivs)
1986
        VGlue u equivs
                            -> VGlueElem (normal ns u) (normal ns us)
1987
         VGlueElem u us
                            -> VUnGlueElem (normal ns u) (normal ns us)
1988
        VUnGlueElem u us
        VUnGlueElemU e u us -> VUnGlueElemU (normal ns e) (normal ns u) (normal ns us)
1989
1990
                            -> VCompU (normal ns a) (normal ns ts)
        VCompU a ts
1991
        VVar x t
                            -> VVar x (normal ns t)
                            -> VFst (normal ns t)
1992
        VFst t
        VSnd t
                            -> VSnd (normal ns t)
1993
1994
        VSplit u t
                            -> VSplit (normal ns u) (normal ns t)
1995
        VApp u v
                            -> VApp (normal ns u) (normal ns v)
1996
        VAppFormula u phi -> VAppFormula (normal ns u) (normal ns phi)
1997
        VId a u v
                            -> VId (normal ns a) (normal ns u) (normal ns v)
1998
        VIdPair u us
                            -> VIdPair (normal ns u) (normal ns us)
1999
        VIdJ a u c d x p
                            -> VIdJ (normal ns a) (normal ns u) (normal ns c)
2000
                                     (normal ns d) (normal ns x) (normal ns p)
2001
                            -> V
2002
2003 instance Normal (C.Nameless a) where
2004
      normal = id
2005
2006 instance Normal Ctxt where
2007
      normal_=id
2008
2009 instance Normal C.Formula where
2010
      normal _ = C.fromDNF . C.dnf
2011
2012 instance Normal a => Normal (Map k a) where
2013
      normal ns = Map.map (normal ns)
2014
2015 instance (Normal a, Normal b) => Normal (a,b) where
      normal ns (u,v) = (normal ns u, normal ns v)
2016
2017
```

```
2018 instance (Normal a, Normal b, Normal c) => Normal (a, b, c) where
2019
       normal ns (u,v,w) = (normal ns u, normal ns v, normal ns w)
2020
2021 instance (Normal a, Normal b, Normal c, Normal d) => Normal (a, b, c, d) where
2022
       normal ns (u,v,w,x) =
2023
         (normal ns u, normal ns v, normal ns w, normal ns x)
2024
2025 instance Normal a => Normal [a] where
      normal ns = map (normal ns)
2026
2027 :::::::::::
2028 Main.hs
2029 :::::::::::
2030 module Main where
2031
2032 import Control.Monad.Reader
2033 import qualified Control. Exception as E
2034 import Data.List
2035 import Data. Time
2036 import System.Directory
2037 import System.FilePath
2038 import System. Environment
2039 import System.Console.GetOpt
2040 import System.Console.Haskeline
2041 import System.Console.Haskeline.History
2042 import Text.Printf
2043
2044 import Exp.Lex
2045 import Exp.Par
2046 import Exp.Print
2047 import Exp.Abs hiding (NoArg)
2048 import Exp.Layout
2049 import Exp.ErrM
2050
2051 -- | CubicalTT syntax
2052 import CTT
2053 -- | Resolver for symbol resolution.
2054 import Resolver
2055 -- | Type checker
2056 import qualified TypeChecker as TC
2057 -- | Evaluator
2058 import qualified Eval as E
2059
2060 type Interpreter a = InputT IO a
2061
2062
2063 -- | Flags
2064 data Flag = Debug | Batch | Help | Version | Time
       deriving (Eq,Show)
2065
2066
2067 options :: [OptDescr Flag]
2068 options = [ Option "d"
                             ["debug"]
                                          (NoArg Debug)
                                                          "run in debugging mode"
               , Option "b"
                              ["batch"]
                                                           "run in batch mode"
2069
                                          (NoArg Batch)
               , Option ""
                              ["help"]
2070
                                                           "print help"
                                          (NoArg Help)
               , Option "-t" ["time"]
2071
                                                           "measure time spent computing"
                                          (NoArg Time)
2072
               , Option ""
                              ["version"] (NoArg Version) "print version number" ]
2073
2074
2075 -- | Version number, welcome message, usage and prompt strings
2076 version, welcome, usage, prompt :: String
2077 \text{ version} = "1.0"
2078 welcome = "cubical, version: " ++ version ++ " (:h for help)\n"
           = "Usage: cubical [options] <file.ctt>\nOptions:"
2079 usage
2080 prompt = "> "
2081
2082 -- | Entrypoint. handle command line arguments. If passed a file, load the file
2083 -- and then enter REPL loop. If not, directly enter REPL loop.
2084 main :: IO ()
2085 \text{ main} = do
2086
      args <- getArgs
2087
       case getOpt Permute options args of
```

```
2088
         (flags, files, [])
                      elem` flags -> putStrLn $ usageInfo usage options
2089
           | Help
           | Version `elem` flags -> putStrLn version
2090
2091
           | otherwise -> case files of
2092
            [] -> do
2093
              putStrLn welcome
2094
              runInputT (settings []) (loop flags [] [] TC.verboseEnv)
2095
            [f] -> do
2096
              putStrLn welcome
2097
              putStrLn $ "Loading " ++ show f
2098
              initLoop flags f emptyHistory
                -> putStrLn $ "Input error: zero or one file expected\n\n" ++
2099
2100
                               usageInfo usage options
         (_,_,errs) -> putStrLn $ "Input error: " ++ concat errs ++ "\n" ++
2101
2102
                                   usageInfo usage options
2103
2104 -- | The main loop
2105 loop :: [Flag] -> FilePath -> [(CTT.Ident,SymKind)] -> TC.TEnv -> Interpreter ()
2106 loop flags f names tenv = do
2107
       input <- getInputLine prompt</pre>
2108
       case input of
                    -> outputStrLn help >> loop flags f names tenv
2109
         Nothing
         Just ":q" -> return ()
2110
         Just ":r" -> getHistory >>= lift . initLoop flags f
2111
         Just (':':'l':' ':str)
2112
           | ' ' `elem` str -> do outputStrLn "Only one file allowed after :l"
2113
2114
                                   loop flags f names tenv
2115
                            -> getHistory >>= lift . initLoop flags str
           | otherwise
         Just (':':'c':'d':' ':str) -> do lift (setCurrentDirectory str)
2116
2117
                                           loop flags f names tenv
2118
         Just ":h" -> outputStrLn help >> loop flags f names tenv
2119
         Just str'
                    ->
2120
           let (msg,str,mod) = case str' of
                   :':'n':' ':str) ->
2121
                 ("NORMEVAL: ",str,E.normal [])
str -> ("EVAL: ",str,id)
2122
2123
2124
           in case pExp (lexer str) of
           Bad err -> outputStrLn ("Parse error: " ++ err) >> loop flags f names tenv
2125
2126
           -- | Resolve the expression
2127
           0k exp ->
             case runResolver $ local (insertIdents names) $ resolveExp exp of
2128
               Left err -> do outputStrLn ("Resolver failed: " ++ err)
2129
2130
                                 loop flags f names tenv
2131
               Right body -> do
2132
                 -- | KEY STEP: type check the expression
                 x <- liftIO $ TC.runInfer tenv body
2133
2134
                 case x of
2135
                   Left err -> do outputStrLn ("Could not type-check: " ++ err)
2136
                                   loop flags f names tenv
2137
                   Right
                            -> do
2138
                     start <- liftIO getCurrentTime</pre>
2139
                     -- | KEY STEP: evaluate the expression.
2140
                     let e = mod $ E.eval (TC.env tenv) body
2141
                     -- | Let's not crash if the evaluation raises an error:
2142
                     liftIO $ catch (putStrLn (msg ++ shrink (show e)))
2143
                                     -- (writeFile "examples/nunivalence3.ctt" (show e))
2144
                                     (\e -> putStrLn ("Exception: " ++
2145
                                                       show (e :: SomeException)))
                     stop <- liftIO getCurrentTime</pre>
2146
2147
                      -- | Compute time and print nicely if `-t` is used.
2148
                     let time = diffUTCTime stop start
2149
                          secs = read (takeWhile (/='.') (init (show time)))
2150
                          rest = read ('0':dropWhile (/='.') (init (show time)))
                          mins = secs `quot` 60
2151
                          sec = printf "%.3f" (fromInteger (secs `rem` 60) + rest :: Float)
2152
                     when (Time `elem` flags) $
2153
                        outputStrLn $ "Time: " ++ show mins ++ "m" ++ sec ++ "s"
2154
2155
                     -- Only print in seconds:
2156
                      -- when (Time `elem` flags) $ outputStrLn $ "Time: " ++ show time
2157
                     loop flags f names tenv
```

```
2158
2159 -- | load file
2160 initLoop :: [Flag] -> FilePath -> History -> IO ()
2161 initLoop flags f hist = do
2162
       -- Parse and type check files
       -- | imports defined below. Load modules. Module defined in ???
2163
2164
       (_,_,mods) <- E.catch (imports True ([],[],[]) f)
2165
                              (\e -> do putStrLn $ unlines $
2166
                                          ("Exception: ":
                                           (takeWhile (/= "CallStack (from HasCallStack):")
2167
2168
                                                          (lines $ show (e :: SomeException))))
2169
                                        return ([],[],[]))
2170
       -- | Translate to TT. resolveModules from from Resolver.hs.
2171
       let res = runResolver $ resolveModules mods
2172
       case res of
2173
         left err
                     -> do
2174
           putStrLn $ "Resolver failed: " ++ err
2175
           runInputT (settings []) (putHistory hist >> loop flags f [] TC.verboseEnv)
2176
         Right (adefs, names) -> do
2177
           -- After resolivng the file check if some definitions were shadowed:
2178
           let ns = map fst names
2179
               uns = nub ns
2180
               dups = ns \setminus uns
2181
           unless (dups == []) $
2182
             putStrLn $ "Warning: the following definitions were shadowed [" ++
                        intercalate ", " dups ++ "]"
2183
2184
           (merr,tenv) <- TC.runDeclss TC.verboseEnv adefs</pre>
2185
           case merr of
             Just err -> putStrLn $ "Type checking failed: " ++ shrink err
2186
             Nothing -> unless (mods == []) $ putStrLn "File loaded."
2187
           if Batch `elem` flags
2188
2189
             then return ()
2190
             else -- Compute names for auto completion
2191
                  runInputT (settings [n | (n,_) <- names])</pre>
2192
                    (putHistory hist >> loop flags f names tenv)
2193
2194
2195 -- | TODO: where is this coming from?!
2196 lexer :: String -> [Token]
2197 lexer = resolveLayout True . myLexer
2198
2199 showTree :: (Show a, Print a) => a -> IO ()
2200 showTree tree = do
       putStrLn  (Abstract Syntax] \ n \ "++ show tree 
2201
       putStrLn $ "\n[Linearized tree]\n\n" ++ printTree tree
2202
2203
2204 -- Used for auto completion
2205 searchFunc :: [String] -> String -> [Completion]
2206 searchFunc ns str = map simpleCompletion $ filter (str `isPrefixOf`) ns
2207
2208 settings :: [String] -> Settings IO
2209 settings ns = Settings
2210
                        = Nothing
      { historyFile
                        = completeWord Nothing " \t^{"} $ return . searchFunc ns
2211
       , complete
2212
       , autoAddHistory = True }
2213
2214
2215 shrink :: String -> String
2216 shrink s = s - if length s > 1000 then take 1000 <math>s + + ... else s
2217
2218
2219
2220 -- | import modules
2221 -- (not ok, loaded, already loaded defs) -> to load ->
          (new not ok, new loaded, new defs)
2223 -- the bool determines if it should be verbose or not
2224 imports :: Bool -> ([String],[String],[Module]) -> String ->
                IO ([String],[String],[Module])
2225
2226 imports v st@(notok,loaded,mods) f
2227
     | f `elem` notok = error ("Looping imports in " ++ f)
```

```
| f `elem` loaded = return st
2228
2229
       | otherwise
                         = do
2230
         b <- doesFileExist f
         when (not b) $ error (f ++ " does not exist")
2231
2232
         let prefix = dropFileName f
2233
         s <- readFile f
2234
         let ts = lexer s
         -- | parse a module and store it.
2235
2236
         -- | check that module name is same as filename.
2237
         case pModule ts of
           Bad s -> error ("Parse failed in " ++ show f ++ "\n" ++ show s)
2238
2239
           Ok mod@(Module (AIdent (_,name)) imp decls) -> do
             let imp ctt = [prefix ++ i ++ ".ctt" | Import (AIdent ( ,i)) <- imp]</pre>
2240
             when (name /= dropExtension (takeFileName f)) $
2241
               error ("Module name mismatch in " ++ show f ++ " with wrong name " ++ "\"" ++
2242
name ++ "\"")
2243
             (notok1,loaded1,mods1) <-</pre>
2244
               foldM (imports v) (f:notok,loaded,mods) imp_ctt
             when v $ putStrLn $ "Parsed " ++ show f ++ " successfully!"
2245
2246
             return (notok,f:loaded1,mods1 ++ [mod])
2247
2248 help :: String
2249 help = "\nAvailable commands:\n" ++
2250
                               infer type and evaluate statement\n" ++
              <statement>
2251
                               normalize statementn" ++
              :n <statement>
2252
                               quit n" ++
              : q
2253
              :l <filename>
                               loads filename (and resets environment before)n'' ++
            п
2254
               :cd <path>
                               change directory to path|n| ++
            п
2255
                               reload\n" ++
               : r
            п
2256
               : h
                               display this message\n"
2257 ::::::::::::
2258 Resolver.hs
2259 :::::::::::
2260 {-# LANGUAGE TupleSections #-}
2261
2262 -- | Convert the concrete syntax into the syntax of cubical TT.
2263 module Resolver(
2264
          Resolver
        , resolveModule
2265
       , resolveModules
2266
       , SymKind(..)
2267
       , runResolver
2268
       , resolveExp
2269
2270
        , insertIdents) where
2271
2272 import Control.Applicative
2273 import Control.Monad
2274 import Control.Monad.Reader
2275 import Control.Monad.Except
2276 import Control.Monad.Identity
2277 import Data.Maybe
2278 import Data.List
2279 import Data.Map (Map,(!))
2280 import qualified Data. Map as Map
2281
2282 import Exp.Abs
2283 import CTT (Ter,Ident,Loc(..),mkApps,mkWheres)
2284 import qualified CTT
2285 import Connections (negFormula,andFormula,orFormula)
2286 import qualified Connections as C
2287
2288 -- | Useful auxiliary functions
2289
2290 -- Applicative cons
2291 (<:>) :: Applicative f => f a -> f [a] -> f [a]
2292 a <:> b = (:) <$> a <*> b
2293
2294 -- Un-something functions
2295 unVar :: Exp -> Maybe Ident
2296 unVar (Var (AIdent (\_,x))) = Just x
```

```
2297 unVar _
                                = Nothing
2298
2299 unWhere :: ExpWhere -> Exp
2300 unWhere (Where e ds) = Let ds e
2301 unWhere (NoWhere e) = e
2302
2303 -- Tail recursive form to transform a sequence of applications
2304 -- App (App (App u v) ...) w into (u, [v, ..., w])
2305 -- (cleaner than the previous version of unApps)
2306 unApps :: Exp -> [Exp] -> (Exp, [Exp])
2307 unApps (App u v) ws = unApps u (v : ws)
2308 unApps u
                      ws = (u, ws)
2309
2310 -- Turns an expression of the form App (... (App id1 id2) ... idn)
2311 -- into a list of idents
2312 appsToIdents :: Exp -> Maybe [Ident]
2313 appsToIdents = mapM unVar . uncurry (:) . flip unApps []
2314
2315 -- Transform a sequence of applications
2316 -- (((u v1) .. vn) phi1) .. phim into (u,[v1,..,vn],[phi1,..,phim])
2317 unAppsFormulas :: Exp -> [Formula]-> (Exp,[Exp],[Formula])
2318 unAppsFormulas (AppFormula u phi) phis = unAppsFormulas u (phi:phis)
2319 unAppsFormulas u phis = (x,xs,phis)
2320
       where (x,xs) = unApps u []
2321
2322 -- Flatten a tele
2323 flattenTele :: [Tele] -> [(Ident,Exp)]
2324 flattenTele tele =
       [ (unAIdent i,typ) | Tele id ids typ <- tele, i <- id:ids ]
2325
2326
2327 -- Flatten a PTele
2328 flattenPTele :: [PTele] -> Resolver [(Ident,Exp)]
2329 flattenPTele []
                                       = return []
2330 flattenPTele (PTele exp typ : xs) = case appsToIdents exp of
2331
       Just ids -> do
2332
         pt <- flattenPTele xs
2333
         return $ map (,typ) ids ++ pt
       Nothing -> throwError "malformed ptele"
2334
2335
2336 -----
2337 -- | Resolver and environment
2338
2339 data SymKind = Variable | Constructor | PConstructor | Name
2340
       deriving (Eq,Show)
2341
2342 -- local environment for constructors
2343 data Env = Env { envModule :: String,
2344
                      variables :: [(Ident,SymKind)] }
2345
       deriving (Eq,Show)
2346
2347 type Resolver a = ReaderT Env (Except String) a
2348
2349 emptyEnv :: Env
2350 emptyEnv = Env "" []
2351
2352 runResolver :: Resolver a -> Either String a
2353 runResolver x = runIdentity $ runExceptT $ runReaderT x emptyEnv
2354
2355 updateModule :: String -> Env -> Env
2356 updateModule mod e = e{envModule = mod}
2357
2358 insertIdent :: (Ident,SymKind) -> Env -> Env
2359 insertIdent (n,var) e
       | n == " " = e
2360
2361
       | otherwise = e{variables = (n,var) : variables e}
2362
2363 insertIdents :: [(Ident,SymKind)] -> Env -> Env
2364 insertIdents = flip $ foldr insertIdent
2365
2366 insertName :: AIdent -> Env -> Env
```

```
2367 insertName (AIdent (x, x)) = insertIdent (x, x)
2368
2369 insertNames :: [AIdent] -> Env -> Env
2370 insertNames = flip $ foldr insertName
2371
2372 insertVar :: Ident -> Env -> Env
2373 insertVar x = insertIdent (x, Variable)
2374
2375 insertVars :: [Ident] -> Env -> Env
2376 insertVars = flip $ foldr insertVar
2377
2378 insertAIdent :: AIdent -> Env -> Env
2379 insertAIdent (AIdent (,x)) = insertIdent (x,Variable)
2380
2381 insertAIdents :: [AIdent] -> Env -> Env
2382 insertAIdents = flip $ foldr insertAIdent
2383
2384 getLoc :: (Int,Int) -> Resolver Loc
2385 getLoc l = Loc <$> asks envModule <*> pure l
2386
2387 unAIdent :: AIdent -> Ident
2388 unAIdent (AIdent (,x)) = x
2390 resolveName :: AIdent -> Resolver C.Name
2391 resolveName (AIdent (l,x)) = do
2392
     modName <- asks envModule
2393
      vars <- asks variables
2394
      case lookup x vars of
2395
        Just Name -> return $ C.Name x
2396
         _ -> throwError $ "Cannot resolve name " ++ x ++ " at position " ++
2397
                           show l ++ " in module " ++ modName
2398
2399 resolveVar :: AIdent -> Resolver Ter
2400 resolveVar (Aldent (l,x)) = do
2401
       modName <- asks envModule
2402
       vars
              <- asks variables
2403
       case lookup x vars of
2404
         Just Variable
                          -> return $ CTT.Var x
2405
         Just Constructor -> return $ CTT.Con x []
2406
         Just PConstructor ->
           throwError $ "The path constructor " ++ x ++ " is used as a" ++
2407
                        " variable at " ++ show l ++ " in " ++ modName ++
2408
                        " (path constructors should have their type in" ++
2409
                        " curly braces as first argument)"
2410
2411
         Just Name
                          ->
           throwError $ "Name " ++ x ++ " used as a variable at position " ++
2412
                        show l ++ " in module " ++ modName
2413
         _ -> throwError $ "Cannot resolve variable " ++ x ++ " at position " ++
2414
2415
                           show l ++ " in module " ++ modName
2416
2417 lam :: (Ident,Exp) -> Resolver Ter -> Resolver Ter
2418 lam (a,t) e = CTT.Lam a <$> resolveExp t <*> local (insertVar a) e
2419
2420 lams :: [(Ident,Exp)] -> Resolver Ter -> Resolver Ter
2421 lams = flip $ foldr lam
2422
2423 plam :: AIdent -> Resolver Ter -> Resolver Ter
2424 plam i e = CTT.PLam (C.Name (unAIdent i)) <$> local (insertName i) e
2425
2426 plams :: [AIdent] -> Resolver Ter -> Resolver Ter
2427 plams [] _ = throwError "Empty plam abstraction"
2428 plams xs e = foldr plam e xs
2429
2430 bind :: (Ter -> Ter) -> (Ident,Exp) -> Resolver Ter -> Resolver Ter
2431 bind f (x,t) e = f < $> lam (x,t) e
2432
2433 binds :: (Ter -> Ter) -> [(Ident,Exp)] -> Resolver Ter -> Resolver Ter
2434 binds f = flip $ foldr $ bind f
2435
2436 resolveApps :: Exp -> [Exp] -> Resolver Ter
```

```
2437 resolveApps x xs = mkApps <$> resolveExp x <*> mapM resolveExp xs
2438
2439 resolveExp :: Exp -> Resolver Ter
2440 resolveExp e = case e of
2441
                     -> return CTT.U
2442
      Var x
                     -> resolveVar x
2443
       App t s
                    -> resolveApps x xs
2444
         where (x,xs) = unApps t [s]
2445
       Sigma ptele b -> do
2446
         tele <- flattenPTele ptele
2447
         binds CTT.Sigma tele (resolveExp b)
2448
       Pi ptele b
                     -> do
         tele <- flattenPTele ptele
2449
2450
         binds CTT.Pi tele (resolveExp b)
2451
                     -> bind CTT.Pi ("_",a) (resolveExp b)
2452
                     -> do
       Lam ptele t
2453
         tele <- flattenPTele ptele
2454
         lams tele (resolveExp t)
2455
       Fst t
                     -> CTT.Fst <$> resolveExp t
2456
       Snd t
                     -> CTT.Snd <$> resolveExp t
2457
       Pair t0 ts
                     -> do
2458
         e <- resolveExp t0
2459
         es <- mapM resolveExp ts
2460
         return $ foldr1 CTT.Pair (e:es)
2461
       Split t brs -> do
2462
         t'
             <- resolveExp t
         brs' <- mapM resolveBranch brs</pre>
2463
2464
         l@(Loc n (i,j)) <- getLoc (case brs of
                                       OBranch (AIdent (1,_)) _ _:_ -> l
2465
2466
                                       PBranch (AIdent (l,_)) _ _ :_ -> l
2467
                                         -> (0,0))
2468
         return $ CTT.Split (n ++ "_L" ++ show i ++ "_C" ++ show j) l t' brs'
2469
       Let decls e -> do
2470
         (rdecls,names) <- resolveDecls decls</pre>
2471
         mkWheres rdecls <$> local (insertIdents names) (resolveExp e)
2472
       Plam is e
                   -> plams is (resolveExp e)
2473
       Hole (HoleIdent (l,_)) -> CTT.Hole <$> getLoc l
2474
       AppFormula t phi ->
2475
         let (x,xs,phis) = unAppsFormulas e []
2476
         in case x of
2477
           PCon n a ->
2478
             CTT.PCon (unAIdent n)  resolveExp a <*> mapM resolveExp xs
2479
                                   <*> mapM resolveFormula phis
2480
             -> CTT.AppFormula <$> resolveExp t <*> resolveFormula phi
2481
       PathP a u v
                     -> CTT.PathP <$> resolveExp a <*> resolveExp u <*> resolveExp v
                     -> CTT.Comp <$> resolveExp u <*> resolveExp v <*> resolveSystem ts
2482
       Comp u v ts
2483
       HComp u v ts -> CTT.HComp <$> resolveExp u <*> resolveExp v <*> resolveSystem ts
2484
       Fill u v ts
                     -> CTT.Fill <$> resolveExp u <*> resolveExp v <*> resolveSystem ts
2485
       Trans u v
                     -> CTT.Comp <$> resolveExp u <*> resolveExp v <*> pure Map.empty
2486
       Glue u ts
                     -> CTT.Glue <$> resolveExp u <*> resolveSystem ts
2487
       GlueElem u ts -> CTT.GlueElem <$> resolveExp u <*> resolveSystem ts
2488
       UnGlueElem u ts -> CTT.UnGlueElem <$> resolveExp u <*> resolveSystem ts
2489
                     -> CTT.Id <$> resolveExp a <*> resolveExp u <*> resolveExp v
       Id a u v
2490
                    -> CTT.IdPair <$> resolveExp u <*> resolveSystem ts
       IdPair u ts
2491
       IdJ a t c d x p -> CTT.IdJ <$> resolveExp a <*> resolveExp t <*> resolveExp c
2492
                                  <*> resolveExp d <*> resolveExp x <*> resolveExp p
       _ -> do
2493
2494
        modName <- asks envModule</pre>
         throwError ("Could not resolve " ++ show e ++ " in module " ++ modName)
2495
2496
2497 resolveWhere :: ExpWhere -> Resolver Ter
2498 resolveWhere = resolveExp . unWhere
2499
2500 resolveSystem :: System -> Resolver (C.System Ter)
2501 resolveSystem (System ts) = do
2502
       ts' <- sequence [ (,) <$> resolveFace alpha <*> resolveExp u
2503
                       | Side alpha u <- ts ]
2504
       let alphas = map fst ts'
2505
       unless (nub alphas == alphas) $
2506
         throwError $ "system contains same face multiple times: " ++
```

```
2507
                      C.showListSystem ts'
2508
       -- Note: the symbols in alpha are in scope in u, but they mean 0 or 1
2509
       return $ Map.fromList ts'
2510
2511 resolveFace :: [Face] -> Resolver C.Face
2512 resolveFace alpha =
2513
      Map.fromList <$> sequence [ (,) <$> resolveName i <*> resolveDir d
                                  | Face i d <- alpha ]
2514
2515
2516 resolveDir :: Dir -> Resolver C.Dir
2517 resolveDir Dir0 = return 0
2518 resolveDir Dir1 = return 1
2520 resolveFormula :: Formula -> Resolver C.Formula
2521 resolveFormula (Dir d)
                                     = C.Dir <$> resolveDir d
2522 resolveFormula (Atom i)
                                     = C.Atom <$> resolveName i
2523 resolveFormula (Neg phi)
                                     = negFormula <$> resolveFormula phi
2524 resolveFormula (Conj phi _ psi) =
         andFormula <$> resolveFormula phi <*> resolveFormula psi
2525
2526 resolveFormula (Disj phi psi)
2527
         orFormula     resolveFormula phi <*> resolveFormula psi
2528
2529 resolveBranch :: Branch -> Resolver CTT.Branch
2530 resolveBranch (OBranch (AIdent (_,lbl)) args e) = do
       re <- local (insertAIdents args) $ resolveWhere e
2532
       return $ CTT.OBranch lbl (map unAIdent args) re
2533 resolveBranch (PBranch (AIdent (_,lbl)) args is e) = do
2534
       re <- local (insertNames is . insertAIdents args) $ resolveWhere e
2535
       let names = map (C.Name . unAIdent) is
2536
       return $ CTT.PBranch lbl (map unAIdent args) names re
2537
2538 resolveTele :: [(Ident,Exp)] -> Resolver CTT.Tele
2539 resolveTele []
                           = return []
2540 resolveTele ((i,d):t) =
       ((i,) <$> resolveExp d) <:> local (insertVar i) (resolveTele t)
2541
2542
2543 resolveLabel :: [(Ident,SymKind)] -> Label -> Resolver CTT.Label
2544 resolveLabel
                   _ (OLabel n vdecl) =
       CTT.OLabel (unAIdent n) <$> resolveTele (flattenTele vdecl)
2545
2546 resolveLabel cs (PLabel n vdecl is sys) = do
       let tele' = flattenTele vdecl
2547
2548
                = map fst tele'
           ts
           names = map (C.Name . unAIdent) is
2549
2550
           n'
                = unAIdent n
                 = delete (n',PConstructor) cs
2551
           cs'
2552
       CTT.PLabel n' <$> resolveTele tele' <*> pure names
2553
                     <*> local (insertNames is . insertIdents cs' . insertVars ts)
2554
                           (resolveSystem sys)
2555
2556 -- Resolve a non-mutual declaration; returns resolver for type and
2557 -- body separately
2558 resolveNonMutualDecl :: Decl -> (Ident,Resolver CTT.Ter
2559
                                      ,Resolver CTT.Ter,[(Ident,SymKind)])
2560 resolveNonMutualDecl d = case d of
       DeclDef (AIdent (\_,f)) tele t body ->
2561
         let tele' = flattenTele tele
2562
                   = binds CTT.Pi tele' (resolveExp t)
2563
             а
2564
                   = lams tele' (local (insertVar f) $ resolveWhere body)
2565
         in (f,a,d,[(f,Variable)])
2566
       DeclData x tele sums -> resolveDeclData x tele sums null
2567
       DeclHData x tele sums ->
2568
         resolveDeclData x tele sums (const False) -- always pick HSum
2569
       DeclSplit (AIdent (l,f)) tele t brs ->
         let tele' = flattenTele tele
2570
             vars = map fst tele'
2571
2572
                   = binds CTT.Pi tele' (resolveExp t)
             а
2573
             d
                   = do
2574
                       loc <- getLoc l
2575
                       ty <- local (insertVars vars) $ resolveExp t</pre>
                       brs' <- local (insertVars (f:vars)) (mapM resolveBranch brs)</pre>
2576
```

```
2577
                        lams tele' (return $ CTT.Split f loc ty brs')
2578
         in (f,a,d,[(f,Variable)])
2579
       DeclUndef (AIdent (l,f)) tele t ->
2580
         let tele' = flattenTele tele
2581
                   = binds CTT.Pi tele' (resolveExp t)
2582
             d
                   = CTT.Undef <$> getLoc l <*> a
2583
         in (f,a,d,[(f,Variable)])
2584
2585 -- Helper function to resolve data declarations. The predicate p is
2586 -- used to decide if we should use Sum or HSum.
2587 resolveDeclData :: AIdent -> [Tele] -> [Label] -> ([(Ident,SymKind)] -> Bool) ->
                         (Ident, Resolver Ter, Resolver Ter, [(Ident, SymKind)])
2588
2589 resolveDeclData (AIdent (l,f)) tele sums p =
2590
       let tele' = flattenTele tele
                 = binds CTT.Pi tele' (return CTT.U)
2591
2592
                 = [ (unAIdent lbl,Constructor) | OLabel lbl _ <- sums ]
           CS
                 = [ (unAIdent lbl,PConstructor) | PLabel lbl _ _ _ <- sums ]</pre>
2593
           pcs
2594
                 = if p pcs then CTT.Sum else CTT.HSum
           SUM
2595
           d = lams tele' $ local (insertVar f) $
                 sum <$> getLoc l <*> pure f
2596
2597
                      <*> mapM (resolveLabel (cs ++ pcs)) sums
2598
       in (f,a,d,(f,Variable):cs ++ pcs)
2599
2600 resolveRTele :: [Ident] -> [Resolver CTT.Ter] -> Resolver CTT.Tele
2601 resolveRTele [] = return []
2602 resolveRTele (i:is) (t:ts) = do
2603
      a <- t
2604
       as <- local (insertVar i) (resolveRTele is ts)</pre>
2605
       return ((i,a):as)
2606
2607 -- Best effort to find the location of a declaration. This implementation
2608 -- returns the location of the first identifier it contains.
2609 findDeclLoc :: Decl -> Resolver Loc
2610 findDeclLoc d = getLoc loc
2611
         where loc = fromMaybe (-1, 0) $ mloc d
2612
               mloc d = case d of
                 DeclDef (AIdent (l, _))
2613
                                                   -> Just l
2614
                 DeclData (AIdent (l, _)) _ _
                                                   -> Just l
                 DeclHData (AIdent (l, _)) _ _ -> Just l
DeclSplit (AIdent (l, _)) _ _ -> Just l
DeclUndef (AIdent (l, _)) _ _ -> Just l
2615
2616
2617
                                                   -> listToMaybe $ mapMaybe mloc ds
2618
                 DeclMutual ds
                 DeclOpaque (AIdent (l, _))
                                                   -> Just l
2619
                 DeclTransparent (AIdent (l, _)) -> Just l
2620
2621
                 DeclTransparentAll
                                                   -> Nothing
2622
2623 -- Resolve a declaration
2624 resolveDecl :: Decl -> Resolver (CTT.Decls,[(Ident,SymKind)])
2625 resolveDecl d = case d of
2626
       DeclMutual decls -> do
2627
         let (fs,ts,bs,nss) = unzip4 $ map resolveNonMutualDecl decls
2628
             ns = concat nss -- TODO: some sanity checks? Duplicates!?
         when (nub (map fst ns) /= concatMap (map fst) nss) $
2629
           throwError ("Duplicated constructor or ident: " ++ show nss)
2630
         as <- resolveRTele fs ts
2631
         -- The bodies know about all the names and constructors in the
2632
2633
         -- mutual block
2634
         ds <- sequence $ map (local (insertIdents ns)) bs</pre>
2635
         let ads = zipWith (\ (x,y) z -> (x,(y,z))) as ds
2636
         l <- findDeclLoc d</pre>
2637
         return (CTT.MutualDecls l ads,ns)
2638
       DeclOpaque i -> do
2639
         resolveVar i
         return (CTT.OpaqueDecl (unAIdent i), [])
2640
2641
       DeclTransparent i -> do
2642
         resolveVar i
2643
         return (CTT.TransparentDecl (unAIdent i), [])
2644
       DeclTransparentAll -> return (CTT.TransparentAllDecl, [])
       _ -> do let (f,typ,body,ns) = resolveNonMutualDecl d
2645
               l <- findDeclLoc d</pre>
2646
```

```
a <- typ
2647
2648
               d <- body
2649
               return (CTT.MutualDecls l [(f,(a,d))],ns)
2650
2651 resolveDecls :: [Decl] -> Resolver ([CTT.Decls],[(Ident,SymKind)])
2652 resolveDecls []
                         = return ([],[])
2653 resolveDecls (d:ds) = do
2654
       (rtd,names) <- resolveDecl d</pre>
2655
       (rds,names') <- local (insertIdents names) $ resolveDecls ds</pre>
       return (rtd : rds, names' ++ names)
2656
2657
2658 resolveModule :: Module -> Resolver ([CTT.Decls],[(Ident,SymKind)])
2659 resolveModule (Module (AIdent (_,n)) _ decls) =
       local (updateModule n) $ resolveDecls decls
2660
2661
2662 resolveModules :: [Module] -> Resolver ([CTT.Decls],[(Ident,SymKind)])
2663 resolveModules []
                               = return ([],[])
2664 resolveModules (mod:mods) = do
       (rmod, names) <- resolveModule mod</pre>
2665
       (rmods,names') <- local (insertIdents names) $ resolveModules mods</pre>
2666
2667
       return (rmod ++ rmods, names' ++ names)
2668 :::::::::::
2669 Setup.hs
2670 :::::::::::
2671 import Distribution. Simple
2672 import Distribution.Simple.Program
2673 import System. Process (system)
2674
2675 main :: IO ()
2676 main = defaultMainWithHooks $ simpleUserHooks {
      hookedPrograms = [bnfc],
2677
2678
       preBuild = \args buildFlags -> do
2679
           _ <- system "bnfc --haskell -d Exp.cf"</pre>
2680
           preBuild simpleUserHooks args buildFlags
2681 }
2682
2683 bnfc :: Program
2684 bnfc = (simpleProgram "bnfc") {
2685
         programFindVersion = findProgramVersion "--version" id
2686
       }
2687 :::::::::::
2688 TypeChecker.hs
2689 :::::::::::
2690 {-# LANGUAGE TupleSections #-}
2691 module TypeChecker(
2692
      runInfer
       , verboseEnv
2693
       , runDeclss
2694
       , env
2695
2696
       , TEnv) where
2697
2698 import Control.Applicative hiding (empty)
2699 import Control.Monad
2700 import Control.Monad.Except
2701 import Control.Monad.Reader
2702 import Data.Map (Map,(!),mapWithKey,assocs,filterWithKey,elems,keys
                      ,intersection,intersectionWith,intersectionWithKey
2703
                      ,toList,fromList)
2704
2705 import qualified Data. Map as Map
2706 import qualified Data.Traversable as T
2707
2708 import CTT
2709 import qualified Connections as C
2710 import qualified Eval as E
2711
2712 -- | Type checking monad
2713 type Typing a = ReaderT TEnv (ExceptT String IO) a
2715 -- | Environment for type checker
2716 data TEnv =
```

```
2717
      TEnv { names :: [String] -- generated names
          , indent :: Int
2718
2719
                    :: Env
           , env
2720
           , verbose :: Bool -- Should it be verbose and print what it typechecks?
           } deriving (Eq)
2721
2722
2723 verboseEnv, silentEnv :: TEnv
2724 verboseEnv = TEnv [] 0 emptyEnv True
2725 silentEnv = TEnv [] 0 emptyEnv False
2726
2727 -- Trace function that depends on the verbosity flag
2728 trace :: String -> Typing ()
2729 \text{ trace s} = do
2730 b <- asks verbose
      when b $ liftIO (putStrLn s)
2731
2732
2733 -----
2734 -- | Functions for running computations in the type checker monad
2736 runTyping :: TEnv -> Typing a -> IO (Either String a)
2737 runTyping env t = runExceptT $ runReaderT t env
2738
2739 runDecls :: TEnv -> Decls -> IO (Either String TEnv)
2740 runDecls tenv d = runTyping tenv $ do
2741
      checkDecls d
2742
      return $ addDecls d tenv
2743
2744 runDeclss :: TEnv -> [Decls] -> IO (Maybe String, TEnv)
2745 runDeclss tenv []
                        = return (Nothing, tenv)
2746 runDeclss tenv (d:ds) = do
2747
     x <- runDecls tenv d
2748
      case x of
2749
        Right tenv' -> runDeclss tenv' ds
2750
        Left s
                  -> return (Just s, tenv)
2751
2752 runInfer :: TEnv -> Ter -> IO (Either String Val)
2753 runInfer lenv e = runTyping lenv (infer e)
2754
2755 --
2756 -- | Modifiers for the environment
2757
2758 addTypeVal :: (Ident,Val) -> TEnv -> TEnv
2759 addTypeVal (x,a) (TEnv ns ind rho v) =
2760
      let w@(VVar n _) = mkVarNice ns x a
      in TEnv (n:ns) ind (upd (x,w) rho) v
2761
2763 addSub :: (C.Name, C.Formula) -> TEnv -> TEnv
2764 addSub iphi (TEnv ns ind rho v) = TEnv ns ind (sub iphi rho) v
2765
2766 addSubs :: [(C.Name, C.Formula)] -> TEnv -> TEnv
2767 addSubs = flip $ foldr addSub
2768
2769 addType :: (Ident,Ter) -> TEnv -> TEnv
2770 addType (x,a) tenv@(TEnv _ _ rho _) = addTypeVal (x,E.eval rho a) tenv
2771
2772 addBranch :: [(Ident, Val)] -> Env -> TEnv -> TEnv
2773 addBranch nvs env (TEnv ns ind rho v) =
2774
      TEnv ([n \mid (\_,VVar \mid n\_) < - nvs] + + ns) ind (upds nvs rho) v
2775
2776 addDecls :: Decls -> TEnv -> TEnv
2777 addDecls d (TEnv ns ind rho v) = TEnv ns ind (def d rho) v
2779 addTele :: Tele -> TEnv -> TEnv
2780 addTele xas lenv = foldl (flip addType) lenv xas
2781
2782 faceEnv :: C.Face -> TEnv -> TEnv
2783 faceEnv alpha tenv = tenv{env=env tenv `C.face` alpha}
2785 ------
2786 -- | Various useful functions
```

```
2787
2788 -- Extract the type of a label as a closure
2789 getLblType :: LIdent -> Val -> Typing (Tele, Env)
2790 getLblType c (Ter (Sum \_ cas) r) = case lookupLabel c cas of
       Just as -> return (as,r)
2792
       Nothing -> throwError ("getLblType: " ++ show c ++ " in " ++ show cas)
2793 getLblType c (Ter (HSum _ _ _ cas) r) = case lookupLabel c cas of
       Just as \rightarrow return (as,\overline{r})
2794
       Nothing -> throwError ("getLblType: " ++ show c ++ " in " ++ show cas)
2796 getLblType c u = throwError ("expected a data type for the constructor '
2797
                                  ++ c ++ " but got " ++ show u)
2798
2799 -- Monadic version of unless
2800 unlessM :: Monad m => m Bool -> m () -> m ()
2801 unlessM mb x = mb >>= flip unless x
2802
2803 mkVars :: [String] -> Tele -> Env -> [(Ident, Val)]
2804 mkVars _ []
                             = []
2805 mkVars ns ((x,a):xas) nu =
2806
       let w@(VVar n _) = mkVarNice ns x (E.eval nu a)
2807
       in (x,w): mkVars (n:ns) xas (upd (x,w) nu)
2808
2809 -- Test if two values are convertible
2810 (===) :: E.Convertible a => a -> a -> Typing Bool
2811 u === v = E.conv <$> asks names <*> pure u <*> pure v
2812
2813 -- eval in the typing monad
2814 evalTyping :: Ter -> Typing Val
2815 evalTyping t = E.eval <$> asks env <*> pure t
2816
2817 -----
            -----
2818 -- | The bidirectional type checker
2819
2820 -- Check that t has type a
2821 check :: Val -> Ter -> Typing ()
2822 check a t = case(a,t) of
       (_,Undef{}) -> return ()
2823
2824
       (_,Hole l) -> do
2825
           rho <- asks env
           let e = unlines (reverse (context0fEnv rho))
2826
2827
           ns <- asks names
           trace $ "\nHole at " ++ show l ++ ":\n\n" ++
2828
                   e ++ replicate 80 '-' ++ "\n" ++ show (E.normal ns a) ++ "\n"
2829
       (_,Con c es) -> do
2830
2831
         (bs,nu) <- getLblType c a
2832
         checks (bs,nu) es
2833
       (VU,Pi f)
                      -> checkFam f
2834
       (VU,Sigma f)
                      -> checkFam f
       (VU,Sum _ _ bs) -> forM_ bs $ \lbl -> case lbl of
  OLabel _ tele -> checkTele tele
  PLabel _ tele is ts ->
2835
2836
2837
           throwError $ "check: no path constructor allowed in " ++ show t
2838
2839
       (VU, HSum _ bs) -> forM_bs $ \bl -> case lbl of
         OLabel _ tele -> checkTele tele
2840
         PLabel _ tele is ts -> do
2841
2842
           checkTele tele
2843
           rho <- asks env
           unless (all ('elem' is) (C.domain ts)) $
2844
2845
             throwError "names in path label C.System" -- TODO
           mapM_ checkFresh is
2846
2847
           let iis = zip is (map C.Atom is)
2848
           local (addSubs iis . addTele tele) $ do
2849
             checkSystemWith ts $ \alpha talpha ->
2850
               local (faceEnv alpha) $
2851
                 -- NB: the type doesn't depend on is
                 check (Ter t rho) talpha
2852
             rho' <- asks env
2853
2854
             checkCompSystem (E.evalSystem rho' ts)
       (VPi va@(Ter (Sum _ _ cas) nu) f,Split _ _ ty ces) -> do
2855
2856
         check VU ty
```

```
2857
         rho <- asks env
2858
         unlessM (a === E.eval rho ty) $ throwError "check: split annotations"
2859
         if map labelName cas == map branchName ces
            then sequence_ [ checkBranch (lbl,nu) f brc (Ter t rho) va
2860
2861
                            | (brc, lbl) <- zip ces cas ]</pre>
2862
            else throwError "case branches does not match the data type"
2863
       (VPi va@(Ter (HSum \_ cas) nu) f,Split \_ ty ces) -> do
2864
         check VU ty
2865
         rho <- asks env
2866
         unlessM (a === E.eval rho ty) $ throwError "check: split annotations"
2867
         if map labelName cas == map branchName ces
2868
            then sequence_ [ checkBranch (lbl,nu) f brc (Ter t rho) va
2869
                              (brc, lbl) <- zip ces cas ]
       else throwError "case branches does not match the data type" (VPi a f,Lam \times a' t) -> do
2870
2871
2872
         check VU a'
2873
         ns <- asks names
2874
         rho <- asks env
2875
         unlessM (a === E.eval rho a') $
2876
           throwError $ "check: lam types don't match"
2877
             ++ "\nlambda type annotation: " ++ show a'
2878
             ++ "\ndomain of Pi: " ++ show a
             ++ "\nnormal form of type: " ++ show (E.normal ns a)
2879
2880
         let var = mkVarNice ns x a
2881
2882
         local (addTypeVal (x,a)) $ check (E.app f var) t
2883
       (VSigma a f, Pair t1 t2) -> do
2884
         check a t1
2885
         v <- evalTyping t1</pre>
2886
         check (E.app f v) t2
2887
       (_,Where e d) -> do
         local (\tenv@TEnv{indent=i} -> tenv{indent=i + 2}) $ checkDecls d
2888
2889
         local (addDecls d) $ check a e
2890
       (VU,PathP a e0 e1) -> do
2891
         (a0,a1) <- checkPLam (constPath VU) a
2892
         check a0 e0
2893
         check al el
       (VPathP p a0 a1,PLam _ e) -> do
2894
2895
         (u0,u1) < - checkPLam p t
2896
         ns <- asks names
2897
         unless (E.conv ns a0 u0 && E.conv ns a1 u1) $
           throwError $ "path endpoints don't match for " ++ show e ++ ", got " ++
2898
                         show (u0,u1) ++ ", but expected " ++ show (a0,a1)
2899
       (VU,Glue a ts) -> do
2900
2901
         check VU a
2902
         rho <- asks env
2903
         checkGlue (E.eval rho a) ts
2904
       (VGlue va ts,GlueElem u us) -> do
2905
         check va u
2906
         vu <- evalTyping u
2907
         checkGlueElem vu ts us
2908
       (VCompU va ves,GlueElem u us) -> do
2909
         check va u
2910
         vu <- evalTyping u
2911
         checkGlueElemU vu ves us
2912
       (VU, Id a a0 a1) -> do
2913
         check VU a
2914
         va <- evalTyping a
2915
         check va a0
2916
         check va al
2917
       (VId va va0 va1, IdPair w ts) -> do
2918
         check (VPathP (constPath va) va0 va1) w
2919
         vw <- evalTyping w
         checkSystemWith ts $ \alpha tAlpha ->
2920
2921
           local (faceEnv alpha) $ do
2922
             check (va `C.face` alpha) tAlpha
             vtAlpha <- evalTyping tAlpha
2923
2924
             unlessM (vw `C.face` alpha === constPath vtAlpha) $
               throwError "malformed eqC"
2925
2926
         rho <- asks env
```

```
2927
         checkCompSystem (E.evalSystem rho ts) -- Not needed
      _ -> do
2928
2929
        v <- infer t
2930
         unlessM (v === a) $
2931
           throwError \ "check conv:\n" ++ show v ++ "\n/=\n" ++ show a
2932
2933 -- Check a list of declarations
2934 checkDecls :: Decls -> Typing ()
                              _ []) = return ()
2935 checkDecls (MutualDecls
2936 checkDecls (MutualDecls l d) = do
2937
       a <- asks env
2938
       let (idents, tele, ters) = (declIdents d, declTele d, declTers d)
2939
       ind <- asks indent
       trace (replicate ind ' ' ++ "Checking: " ++ unwords idents)
2940
2941
       checkTele tele
2942
       local (addDecls (MutualDecls l d)) $ do
2943
         rho <- asks env
         checks (tele, rho) ters
2944
2945 checkDecls (OpaqueDecl )
                                    = return ()
2946 checkDecls (Transparent\overline{D}ecl _) = return ()
2947 checkDecls TransparentAllDecl = return ()
2948
2949 -- Check a telescope
2950 checkTele :: Tele -> Typing ()
2951 checkTele []
                           = return ()
2952 checkTele ((x,a):xas) = do
2953
       check VU a
2954
       local (addType (x,a)) $ checkTele xas
2955
2956 -- Check a family
2957 checkFam :: Ter -> Typing ()
2958 checkFam (Lam x a b) = do
2959
       check VU a
2960
       local (addType (x,a)) $ check VU b
2961 checkFam x = throwError $ "checkFam: " ++ show x
2962
2963 -- Check that a C.System is compatible
2964 checkCompSystem :: C.System Val -> Typing ()
2965 checkCompSystem vus = do
2966
      ns <- asks names
2967
       unless (E.isCompSystem ns vus)
         (throwError $ "Incompatible System " ++ C.showSystem vus)
2968
2969
2970 -- Check the values at corresponding faces with a function, assumes
2971 -- systems have the same faces
2972 checkSystemsWith :: C.System a -> C.System b -> (C.Face -> a -> b -> Typing c) ->
2973
                         Typing ()
2974 checkSystemsWith us vs f = sequence_ $ elems $ intersectionWithKey f us vs
2975
2976 -- Check the faces of a C.System
2977 checkSystemWith :: C.System a -> (C.Face -> a -> Typing b) -> Typing ()
2978 checkSystemWith us f = sequence_ $ elems $ mapWithKey f us
2979
2980 -- Check a glueElem
2981 checkGlueElem :: Val -> C.System Val -> C.System Ter -> Typing ()
2982 checkGlueElem vu ts us = do
2983
       unless (keys ts == keys us)
2984
         (throwError ("Keys don't match in " ++ show ts ++ " and " ++ show us))
2985
       rho <- asks env
       checkSystemsWith ts us
2986
2987
         (\alpha vt u -> local (faceEnv alpha) $ check (E.equivDom vt) u)
2988
       let vus = E.evalSystem rho us
2989
       checkSystemsWith ts vus (\alpha vt vAlpha ->
         unlessM (E.app (E.equivFun vt) vAlpha === (vu `C.face` alpha)) $
2990
           throwError $ "Image of glue component " ++ show vAlpha ++
2991
                        " doesn't match " ++ show vu)
2992
2993
       checkCompSystem vus
2994
2995 -- Check a glueElem against VComp _ ves
2996 checkGlueElemU :: Val -> C.System Val -> C.System Ter -> Typing ()
```

```
2997 checkGlueElemU vu ves us = do
2998
       unless (keys ves == keys us)
2999
         (throwError ("Keys don't match in " ++ show ves ++ " and " ++ show us))
3000
       rho <- asks env
3001
       checkSystemsWith ves us
3002
         (\alpha ve u -> local (faceEnv alpha) $ check (ve E.@@ C.One) u)
3003
       let vus = E.evalSystem rho us
3004
       checkSystemsWith ves vus (\alpha ve vAlpha ->
3005
         unlessM (E.eqFun ve vAlpha === (vu `C.face` alpha)) $
3006
           throwError $ "Transport of glueElem (for compU) component " ++ show vAlpha ++
                        " doesn't match " ++ show vu)
3007
3008
       checkCompSystem vus
3009
3010 checkGlue :: Val -> C.System Ter -> Typing ()
3011 checkGlue va ts = do
       checkSystemWith ts (\alpha tAlpha -> checkEquiv (va `C.face` alpha) tAlpha)
3012
3013
       rho <- asks env
3014
       checkCompSystem (E.evalSystem rho ts)
3015
3016 -- An iso for a type b is a five-tuple: (a,f,g,s,t) where
3017 -- a : U
3018 -- f : a -> b
3019 -- g : b -> a
3020 -- s : forall (y : b), f (g y) = y
3021 -- t : forall (x : a), g (f x) = x
3022 mkIso :: Val -> Val
3023 mkIso vb = E.eval rho $
3024
       Sigma $ Lam "a" U $
       Sigma $ Lam "f" (Pi (Lam "_" a b)) $
Sigma $ Lam "g" (Pi (Lam "_" b a)) $
3025
3026
3027
       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)
3028
       where [a,b,f,g,x,y] = map Var ["a","b","f","g","x","y"]
3029
3030
             rho = upd ("b",vb) emptyEnv
3031
3032 -- An equivalence for a type a is a triple (t,f,p) where
3033 -- t : U
3034 -- f : t -> a
3035 -- p : (x : a) -> isContr ((y:t) * Id a x (f y))
3036 -- with isContr c = (z : c) * ((z' : C) -> Id c z z')
3037 mkEquiv :: Val -> Val
3038 mkEquiv va = E.eval rho $
       Sigma $ Lam "t" U $
3039
       Sigma $ Lam "f" (Pi (Lam "_" t a)) $
3040
       Pi (Lam "x" a $ iscontrfib)
3041
       where [a,b,f,x,y,s,t,z] = map Var ["a","b","f","x","y","s","t","z"]
3042
3043
             rho = upd ("a",va) emptyEnv
3044
             fib = Sigma $ Lam "y" t (PathP (PLam (C.Name "_") a) x (App f y))
3045
             iscontrfib = Sigma $ Lam "s" fib $
                          Pi $ Lam "z" fib $ PathP (PLam (C.Name "_") fib) s z
3046
3047
3048 checkEquiv :: Val -> Ter -> Typing ()
3049 checkEquiv va equiv = check (mkEquiv va) equiv
3051 checkIso :: Val -> Ter -> Typing ()
3052 checkIso vb iso = check (mkIso vb) iso
3053
3054 checkBranch :: (Label,Env) -> Val -> Branch -> Val -> Val -> Typing ()
3055 checkBranch (OLabel _ tele,nu) f (OBranch c ns e) _ _ = do
3056
       ns' <- asks names
3057
       let us = map snd $ mkVars ns' tele nu
3058
       local (addBranch (zip ns us) nu) $ check (E.app f (VCon c us)) e
3059 checkBranch (PLabel _ tele is ts,nu) f (PBranch c ns js e) g va = do
3060
       ns' <- asks names
3061
       -- mapM_ checkFresh js
3062
       let us
              = mkVars ns' tele nu
           vus = map snd us
3063
           js'
               = map C.Atom js
3064
           vts = E.evalSystem (subs (zip is js') (upds us nu)) ts
3065
3066
           vgts = intersectionWith E.app (C.border g vts) vts
```

```
3067
       local (addSubs (zip js js') . addBranch (zip ns vus) nu) $ do
3068
         check (E.app f (VPCon c va vus js')) e
3069
         ve <- evalTyping e -- TODO: combine with next two lines?
3070
         let veborder = C.border ve vts
3071
         unlessM (veborder === vgts) $
           throwError $ "Faces in branch for " ++ show c ++ " don't match:"
3072
3073
                        ++ "\ngot\n" ++ C.showSystem veborder ++ "\nbut expected\n"
3074
                        ++ C.showSystem vgts
3075
3076 checkFormula :: C.Formula -> Typing ()
3077 checkFormula phi = do
3078
       rho <- asks env
3079
       let dom = domainEnv rho
       unless (all (`elem` dom) (C.support phi)) $
3080
         throwError $ "checkFormula: " ++ show phi
3081
3082
3083 checkFresh :: C.Name -> Typing ()
3084 checkFresh i = do
3085
       rho <- asks env
3086
       when (i `elem` C.support rho)
3087
         (throwError $ show i ++ " is already declared")
3088
3089 -- Check that a term is a PLam and output the source and target
3090 checkPLam :: Val -> Ter -> Typing (Val, Val)
3091 checkPLam v (PLam i a) = do
3092
      rho <- asks env
3093
       -- checkFresh i
3094
       local (addSub (i,C.Atom i)) $ check (v E.@@ i) a
3095
       return (E.eval (sub (i,C.Dir 0) rho) a, E.eval (sub (i,C.Dir 1) rho) a)
3096 checkPLam v t = do
3097
      vt <- infer t
3098
       case vt of
3099
         VPathP a a0 a1 -> do
3100
           unlessM (a === v) $ throwError (
3101
             "checkPLam\n" ++ show v ++ "\n/=\n" ++ show a)
3102
           return (a0,a1)
         _ -> throwError $ show vt ++ " is not a path"
3103
3104
3105 -- Return C.System such that:
          rhoalpha |- p_alpha : Id (va alpha) (t0 rhoalpha) ualpha
3106 --
3107 -- Moreover, check that the C.System ps is compatible.
3108 checkPLamSystem :: Ter -> Val -> C.System Ter -> Typing (C.System Val)
3109 checkPLamSystem t0 va ps = do
3110
       rho <- asks env
3111
       v <- T.sequence $ mapWithKey (\alpha pAlpha ->
3112
         local (faceEnv alpha) $ do
           rhoAlpha <- asks env
3113
3114
           (a0,a1) <- checkPLam (va `C.face` alpha) pAlpha</pre>
3115
           unlessM (a0 === E.eval rhoAlpha t0) $
             throwError $ "Incompatible C.System " ++ C.showSystem ps ++
3116
3117
                          ", component\n " ++ show pAlpha ++
                          "\nincompatible with\n " ++ show t0 ++
3118
                          "\na0 = " ++ show a0 ++
3119
                          "\nt0alpha = " ++ show (E.eval rhoAlpha t0) ++
3120
                          "\nva = " ++ show va
3121
3122
           return a1) ps
3123
       checkCompSystem (E.evalSystem rho ps)
3124
       return v
3125
3126 checks :: (Tele,Env) -> [Ter] -> Typing ()
3127 checks ([],_)
                           []
                                  = return ()
3128 checks ((x,a):xas,nu) (e:es) = do
3129
       check (E.eval nu a) e
       v' <- evalTyping e
3130
       checks (xas,upd (x,v') nu) es
3131
3132 checks
       throwError "checks: incorrect number of arguments"
3133
3134
3135 -- | infer the type of e
3136 infer :: Ter -> Typing Val
```

```
3137 infer e = case e of
3138
                    -> return VU -- U : U
       U
3139
       Var n
                    -> E.lookType n <$> asks env
3140
       App t u -> do
3141
         c <- infer t
3142
         case c of
3143
           VPi a f -> do
3144
             check a u
3145
             v <- evalTyping u</pre>
3146
              return $ E.app f v
3147
                    -> throwError $ show c ++ " is not a product"
       Fst \overline{t} \rightarrow do
3148
3149
         c <- infer t
3150
         case c of
3151
           VSigma a f -> return a
                       -> throwError $ show c ++ " is not a sigma-type"
3152
       Snd \overline{t} \rightarrow do
3153
3154
         c <- infer t
3155
         case c of
3156
            VSigma a f -> do
3157
             v <- evalTyping t</pre>
3158
              return $ E.app f (E.fstVal v)
3159
                       -> throwError $ show c ++ " is not a sigma-type"
3160
       Where t d -> do
3161
         checkDecls d
3162
         local (addDecls d) $ infer t
3163
       UnGlueElem e _ -> do
3164
         t <- infer e
3165
         case t of
3166
          VGlue a _ -> return a
            -> throwError (show t ++ " is not a Glue")
3167
3168
       AppFormula e phi -> do
3169
         checkFormula phi
3170
         t <- infer e
3171
         case t of
                         -> return $ a E.@@ phi
3172
           VPathP a
3173
            -> throwError (show e ++ " is not a path")
3174
       Comp a t0 ps -> do
          (va0, va1) <- checkPLam (constPath VU) a
3175
3176
         va <- evalTyping a
3177
         check va0 t0
3178
         checkPLamSystem t0 va ps
3179
         return val
3180
       HComp a u0 us -> do
3181
         check VU a
3182
         va <- evalTyping a
3183
         check va u0
3184
         checkPLamSystem u0 (constPath va) us
3185
         return va
3186
       Fill a t0 ps -> do
3187
         (va0, va1) <- checkPLam (constPath VU) a
3188
         va <- evalTyping a
3189
         check va0 t0
3190
         checkPLamSystem t0 va ps
3191
         vt <- evalTyping t0
3192
         rho <- asks env
3193
         let vps = E.evalSystem rho ps
3194
         return (VPathP va vt (E.compLine va vt vps))
3195
       PCon c a es phis -> do
3196
         check VU a
3197
         va <- evalTyping a</pre>
3198
         (bs,nu) <- getLblType c va</pre>
3199
         checks (bs,nu) es
3200
         mapM_ checkFormula phis
3201
         return va
3202
       IdJ a u c d x p \rightarrow do
3203
         check VU a
3204
         va <- evalTyping a
         check va u
3205
3206
         vu <- evalTyping u
```

```
3207
         let refu = VIdPair (constPath vu) $ C.mkSystem [(C.eps,vu)]
3208
         rho <- asks env
3209
         let z = Var "z"
              ctype = E.eval rho $ Pi $ Lam "z" a $ Pi $ Lam " " (Id a u z) U
3210
3211
         check ctype c
3212
         vc <- evalTyping c
3213
         check (E.app (E.app vc vu) refu) d
3214
         check va x
3215
         vx <- evalTyping x</pre>
3216
         check (VId va vu vx) p
3217
         vp <- evalTyping p</pre>
3218
         return (E.app (E.app vc vx) vp)
       _ -> throwError ("infer " ++ show e)
3219
3220
3221 -- Not used since we have U : U
3222 --
3223 -- (=?=) :: Typing Ter -> Ter -> Typing ()
3224 -- m =?= s2 = do
3225 --
          s1 <- m
3226 --
          unless (s1 == s2) $ throwError (show s1 ++ " =/= " ++ show s2)
3227 --
3228 -- checkTs :: [(String, Ter)] -> Typing ()
3229 -- checkTs [] = return ()
3230 -- checkTs ((x,a):xas) = do
3231 --
          checkType a
3232 --
          local (addType (x,a)) (checkTs xas)
3233 --
3234 -- checkType :: Ter -> Typing ()
3235 -- checkType t = case t of
3236 --
                          -> return ()
         11
3237 --
          Pi \ a \ (Lam \ x \ b) \ -> \ do
3238 --
          checkType a
3239 --
            local (addType (x,a)) (checkType b)
3240 --
            -> infer t =?= U
3241 :::::::::::
3242 Exp.cf
3243 :::::::::::
3244 entrypoints Module, Exp ;
3245
3246 comment "--"
3247 comment "{-" "-}";
3248
3249 layout "where", "let", "split", "mutual", "with";
3250 layout stop "in" ;
3251 -- Do not use layout toplevel as it makes pExp fail!
3252
3253 Module.
               Module ::= "module" AIdent "where" "{" [Imp] [Decl] "}" ;
3255 Import.
                Imp ::= "import" AIdent ;
3256 separator Imp ";";
3257
                           Decl ::= AIdent [Tele] ":" Exp "=" ExpWhere ;
3258 DeclDef.
                           Decl ::= "data" AIdent [Tele] "=" [Label] ;
3259 DeclData.
                           Decl ::= "hdata" AIdent [Tele] "=" [Label] ;
Decl ::= AIdent [Tele] ":" Exp "=" "split" "{" [Branch] "}" ;
3260 DeclHData.
3261 DeclSplit.
3262 DeclUndef.
                           Decl ::= AIdent [Tele] ":" Exp "=" "undefined" ;
                           Decl ::= "mutual" "{" [Decl] "}" ;
3263 DeclMutual.
                           Decl ::= "opaque" AIdent ;
3264 DeclOpaque.
3265 DeclTransparent.
                           Decl ::= "transparent" AIdent ;
                           Decl ::= "transparent_all" ;
Decl ";" ;
3266 DeclTransparentAll.
3267 separator
3268
                ExpWhere ::= Exp "where" "{" [Decl] "}" ;
3269 Where.
3270 NoWhere. ExpWhere ::= Exp ;
3271
                    Exp ::= "let" "{" [Decl] "}" "in" Exp ;
3272 Let.
                    Exp ::= "\\" [PTele] "->" Exp ;
3273 Lam.
                    Exp ::= "<" [AIdent] ">" Exp ;
Exp ::= "split@" Exp "with" "{" [Branch] "}" ;
3274 PLam.
3275 Split.
                    Exp1 ::= Exp2 "->" Exp1 ;
3276 Fun.
```

```
Exp1 ::= [PTele] "->" Exp1 ;
3277 Pi.
                   Exp1 ::= [PTele] "*" Exp1 ;
3278 Sigma.
                   Exp2 ::= Exp2 "@" Formula ;
3279 AppFormula.
3280 App.
                   Exp2 ::= Exp2 Exp3 ;
                   Exp3 ::= "PathP" Exp4 Exp4 Exp4 ;
3281 PathP.
3282 Comp.
                   Exp3 ::= "comp" Exp4 Exp4 System ;
3283 HComp.
                   Exp3 ::= "hComp" Exp4 Exp4 System ;
                   Exp3 ::= "transport" Exp4 Exp4 ;
3284 Trans.
                   Exp3 ::= "fill" Exp4 Exp4 System ;
3285 Fill.
                   Exp3 ::= "Glue" Exp4 System ;
3286 Glue.
                   Exp3 ::= "qlue" Exp4 System ;
3287 GlueElem.
                   Exp3 ::= "unglue" Exp4 System ;
3288 UnGlueElem.
                   Exp3 ::= "Id" Exp4 Exp4 Exp3 ;
3289 Id.
                   Exp3 ::= "idC" Exp4 System ;
3290 IdPair.
                   Exp3 ::= "idJ" Exp4 Exp4 Exp4 Exp4 Exp4 Exp4 ;
3291 IdJ.
                   Exp4 ::= Exp4 ".1" ;
3292 Fst.
                   Exp4 ::= Exp4 ".2"
3293 Snd.
                   Exp5 ::= "(" Exp "," [Exp] ")" ;
3294 Pair.
3295 Var.
                   Exp5 ::= AIdent
                   Exp5 ::= AIdent "{" Exp "}" ; -- c{T A B} x1 x2 @ phi
3296 PCon.
3297 U.
                   Exp5 ::= "U" :
3298 Hole.
                   Exp5 ::= HoleIdent ;
3299 coercions Exp 5;
3300 separator nonempty Exp ",";
                 Dir ::= "0"
3302 Dir0.
3303 Dir1.
                 Dir ::= "1" ;
3304
3305 System.
                 System ::= "[" [Side] "]" ;
3306
               Face ::= "(" AIdent "=" Dir ")";
3307 Face.
3308 separator Face "";
3309
3310 Side.
             Side ::= [Face] "->" Exp ;
3311 separator Side ",";
3312
               Formula ::= Formula "\\/" Formula1 ;
3313 Disj.
               Formula1 ::= Formula1 CIdent Formula2 ;
3314 Conj.
               Formula2 ::= "-" Formula2 ;
3315 Neg.
3316 Atom.
               Formula2 ::= AIdent ;
3317 Dir.
               Formula2 ::= Dir ;
3318 coercions Formula 2;
3319
3320 -- Branches
                Branch ::= AIdent [AIdent] "->" ExpWhere ;
3321 OBranch.
3322 -- TODO: better have ... @ i @ j @ k -> ... ?
                Branch ::= AIdent [AIdent] "@" [AIdent] "->" ExpWhere ;
3323 PBranch.
3324 separator Branch ";";
3325
3326 -- Labelled sum alternatives
3327 OLabel.
             Label ::= AIdent [Tele] ;
               Label ::= AIdent [Tele] "<" [AIdent] ">" System ;
3328 PLabel.
3329 separator Label "|";
3330
3331 -- Telescopes
               Tele ::= "(" AIdent [AIdent] ":" Exp ")" ;
3332 Tele.
3333 terminator Tele "";
3335 -- Nonempty telescopes with Exp:s, this is hack to avoid ambiguities
3336 -- in the grammar when parsing Pi
               PTele ::= "(" Exp ":" Exp ")" ;
3337 PTele.
3338 terminator nonempty PTele "";
3339
3340 position token AIdent ('_')|(letter)(letter|digit|'\''|'_')*|('!')(digit)* ;
3341 separator Aldent "";
3342
3343 token CIdent '/''\\';
3344
3345 position token HoleIdent '?';
```