djinn-th

Claude Heiland-Allen

2010–2015

Contents

1 djinn-th.cabal

```
Name:                djinn-th
Version:             0.0.1.1
Synopsis:            Generate executable Haskell code from a type
Description:         Djinn uses a theorem prover for intuitionistic
                     propositional logic to generate a Haskell
                     expression when given a type.
                     .
                     Djinn-TH uses Template Haskell to turn this
                     expression into executable code.

Homepage:            http://code.mathr.co.uk/djinn-th
License:             BSD3
License-file:        LICENSE
Author:              Claude Heiland-Allen
Maintainer:          claude@mathr.co.uk
Category:            Language
Build-type:          Simple

Cabal-version:       >=1.2

Library
  Hs-source-dirs:    src
  Exposed-modules:   Language.Haskell.Djinn
  Other-modules:     Language.Haskell.Djinn.LJT, Language.Haskell.Djinn.↙
      ↳ LJTFormula, Language.Haskell.Djinn.HTypes
  Build-depends:     base >= 4 && < 5, template-haskell, containers >= 0.3 && < ↙
      ↳ 0.6, logict >= 0.4 && < 0.7
  GHC-options:       -Wall
  GHC-prof-options:  -prof -auto-all
```

2 .gitignore

```
dist
```

3 LICENSE

```
Copyright (c) 2005 Lennart Augustsson, Thomas Johnsson
    Chalmers University of Technology
All rights reserved.

This code is derived from software written by Lennart Augustsson
(lennart@augustsson.net).

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. None of the names of the copyright holders may be used to endorse
   or promote products derived from this software without specific
   prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN
IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

*** End of disclaimer. ***
```

4 Setup.hs

```
import Distribution.Simple
main = defaultMain
```

5 src/Language/Haskell/Djinn.hs

```
{-# LANGUAGE TemplateHaskell, ScopedTypeVariables #-}
-----------------------------------------------------------------------------
-- |
-- Module      :  Language.Haskell.Djinn
-- License     :  BSD-style (see the accompanying LICENSE file)
--
-- Maintainer  :  claude@mathr.co.uk
-- Stability   :  experimental
-- Portability :  non-portable (template-haskell)
```

```
10  --
    -- Djinn uses a theorem prover for intuitionistic propositional logic to
    -- generate a Haskell expression when given a type. Djinn-TH uses Template
    -- Haskell to turn this expression into executable code.
    --
15  -- Based mostly on <http://hackage.haskell.org/package/djinn>.
    --
    -- Using Language.Haskell.Djinn generally requires:
    --
    -- @&#x7B;-&#x23; LANGUAGE TemplateHaskell, ScopedTypeVariables &#x23;-&#x7D;@
20  --
    ------------------------------------------------------------------------------

    --
    -- Modified to use TemplateHaskell by Claude Heiland-Allen, 2010
25  --
    -- Copyright (c) 2005 Lennart Augustsson
    -- See LICENSE for licensing details.
    --
    module Language.Haskell.Djinn (
30    djinn,  -- :: Q Type -> Q Exp
      djinns, -- :: Q Type -> Q Exp
      djinnD, -- :: String -> Q Type -> Q [Dec]
      djinnsD -- :: String -> Q Type -> Q [Dec]
      ) where
35
    import Data.List (nub, sortBy)
    import Data.Ord (comparing)
    import Data.Ratio ((%))
    import Data.Set (Set, empty, singleton, union, toList)
40  import Language.Haskell.TH (
      Name, Type(..), Dec(..), Pat(..), Exp(..), Body(..), Clause(..),
      Match(..), Info(..), Con(..), TyVarBndr(..), Q,
      newName, mkName, tupleTypeName, tupleDataName, reify, pprint, report)
    import Control.Monad (forM)
45
    import Language.Haskell.Djinn.HTypes (
      HType(..), HPat(..), HExpr(..), HClause(..), HEnvironment,
      termToHClause, hTypeToFormula, getBinderVars)
    import Language.Haskell.Djinn.LJT (prove)
50
    getConTs :: Type -> Set Name
    getConTs (ForallT _ _ t) = getConTs t
    getConTs (ConT name)     = singleton name
    getConTs (AppT t1 t2)    = getConTs t1 `union` getConTs t2
55  getConTs (TupleT n)      = singleton (tupleTypeName n)
    getConTs _               = empty

    hType :: Type -> HType
    hType (TupleT 0) = HTTuple []
60  hType (TupleT 1)         = error $ "djinn: 1-tuple should not exist"
    -- FIXME kludge for now to handle small tuples...
    -- FIXME kludge to handle GHC's tuple stuff
    hType (AppT (AppT ArrowT t1) t2) = HTArrow (hType t1) (hType t2)
    hType (AppT (AppT (TupleT 2) t1) t2) = HTTuple (map hType [t1, t2])
65  hType (AppT (AppT (ConT   c) t1) t2) | c == tupleTypeName 2 = HTTuple (map hType↙
        ↳ [t1, t2])
```

```
hType (AppT (AppT (AppT (TupleT 3) t1) t2) t3) = HTTuple (map hType [t1, t2, t3↙
    ↳ ])
hType (AppT (AppT (AppT (ConT   c) t1) t2) t3) | c == tupleTypeName 3 = HTTuple ↙
    ↳ (map hType [t1, t2, t3])
hType (AppT (AppT (AppT (AppT (TupleT 4) t1) t2) t3) t4) = HTTuple (map hType [↙
    ↳ t1, t2, t3, t4])
hType (AppT (AppT (AppT (AppT (ConT   c) t1) t2) t3) t4) | c == tupleTypeName 4 ↙
    ↳ = HTTuple (map hType [t1, t2, t3, t4])
hType (AppT (AppT (AppT (AppT (AppT (TupleT 5) t1) t2) t3) t4) t5) = HTTuple (↙
    ↳ map hType [t1, t2, t3, t4, t5])
hType (AppT (AppT (AppT (AppT (AppT (ConT   c) t1) t2) t3) t4) t5) | c == ↙
    ↳ tupleTypeName 5 = HTTuple (map hType [t1, t2, t3, t4, t5])
hType (TupleT n) | n > 5 = error $ "djinn: " ++ show n ++ "-tuple not yet ↙
    ↳ supported (max 5)"
hType (AppT t1 t2) = HTApp (hType t1) (hType t2)
hType (ForallT _ _ t) = hType t
hType (VarT v) = HTVar v
hType (ConT n) = HTCon n
hType t = error $ "djinn: unimplemented in hType: " ++ pprint t

-- two mutually recursive functions chase down all data/type defs

environment :: Type -> Q HEnvironment
environment = fmap concat . mapM environment1 . toList . getConTs

environment1 :: Name -> Q HEnvironment
environment1 name = do
  info <- reify name
  case info of
   ClassI _dec -> fail $ "djinn: unexpected ClassI"
   ClassOpI _n _t _c _fx -> fail $ "djinn: unexpected ClassOpI"
   TyConI dec -> do
    case dec of
     DataD _cxt dName dVars dCtors _derivs -> do
      dTypes <- forM dCtors $ \(NormalC cName cFields) -> do
        let cTypes = map (hType . snd) cFields
        cEnv <- mapM (environment . snd) cFields
        return ((cName, cTypes), cEnv)
      return $ [(dName, (map binderName dVars, HTUnion (map fst dTypes)))]
            ++ (concat . concatMap snd $ dTypes)
     TySynD tName tVars tType -> do
      es <- environment tType
      return $ [(tName, (map binderName tVars, hType tType))] ++ es
     x -> fail $ "djinn: unexpected TyConI " ++ show x
   PrimTyConI n _ar _l -> fail $ "djinn: unexpected PrimTyConI " ++ show n
   DataConI _n _t _tn _fx -> fail $ "djinn: unexpected DataConI"
   VarI _n _t _mdec _fx -> fail $ "djinn: unexpected VarI"
   TyVarI _tvName _tvType -> fail $ "djinn: unexpected TyVarI"

binderName :: TyVarBndr -> Name
binderName (PlainTV n) = n
binderName (KindedTV n _k) = n

pat :: HPat -> Pat
pat (HPVar s) = VarP s
pat (HPTuple ps) = TupP (map pat ps)
pat (HPAt s p) = AsP s (pat p)
```

```
pat (HPCon c) = ConP c []
pat (HPApply p q) = let ConP c ps = pat p in ConP c (ps ++ [pat q])

expr :: HExpr -> Exp
expr (HELam ps e) = LamE (map pat ps) (expr e)
expr (HEApply e f) = AppE (expr e) (expr f)
expr (HECon c) = ConE c
expr (HEVar v) = VarE v
expr (HETuple es) = foldl AppE (ConE (tupleDataName (length es))) (map expr es)
expr (HECase e ms) = CaseE (expr e) (map case1 ms)
  where case1 (p, f) = Match (pat p) (NormalB $ expr f) []

djinn0 :: Bool -> Maybe String -> Type -> Q Exp
djinn0 multi mStr typ = do
  syns <- environment typ
  name <- case mStr of
    Nothing -> newName "djinn"
    Just s -> return $ mkName s
  let form = hTypeToFormula syns (hType typ)
  ps <- (nub . map snd . sortBy (comparing fst) . map (f name)) `fmap` (prove ↙
      ↳ multi [] form)
  if multi
   then return $ ListE (map g ps)
   else case  ps of
    ps'@(p:_:_) -> do
      report False $ "djinn: " ++ show (length ps') ++ " options for: " ++ show ↙
          ↳ name ++ " :: " ++ pprint typ
      return $ g p
    [p] -> return $ g p
    [] -> do
      report True $ "djinn: cannot realize: " ++ show name ++ " :: " ++ pprint ↙
          ↳ typ
      x <- newName "djinnError"
      return $ LetE [ValD (VarP x) (NormalB (VarE x)) [] ] (VarE x)
  where
    f name p  = let c = termToHClause name p
                    bvs = getBinderVars c
                    r = if null bvs then (0, 0) else (length (filter (== ↙
                        ↳ underscore) bvs) % length bvs, length bvs)
                in  (r, c)
    g (HClause _ pats body) = let e = expr (HELam pats body) in wilderE e

underscore :: Name
underscore = mkName "_"

wilder :: Pat -> Pat
wilder l@(LitP _) = l
wilder (VarP n) | n == underscore = WildP
wilder (TupP ps) = TupP (map wilder ps)
wilder (ConP n ps) = ConP n (map wilder ps)
wilder (InfixP p1 n p2) = InfixP (wilder p1) n (wilder p2)
wilder (TildeP p) = TildeP (wilder p)
wilder (AsP n p) | n == underscore = wilder p
                 | otherwise = AsP n (wilder p)
--wilder (RecP n fs) = error $ "djinn: field patterns not yet implemented"
wilder (ListP ps) = ListP (map wilder ps)
wilder (SigP p t) = SigP (wilder p) t
```

```
    wilder p = p

    wilderE :: Exp -> Exp
    wilderE (AppE e f) = AppE (wilderE e) (wilderE f)
    wilderE (InfixE me o mf) = InfixE (fmap wilderE me) (wilderE o) (fmap wilderE mf↙
        ↳ )
    wilderE (LamE ps e) = LamE (map wilder ps) (wilderE e)
    wilderE (TupE es) = TupE (map wilderE es)
    wilderE (CondE e f g) = CondE (wilderE e) (wilderE f) (wilderE g)
    wilderE (LetE ds e) = LetE (map wilderD ds) (wilderE e)
    wilderE (CaseE e ms) = CaseE (wilderE e) (map wilderM ms)
    -- DoE [Stmt]                           -- { do { p <- e1; e2 }  }
    -- CompE [Stmt]                         -- { [ (x,y) | x <- xs, y <- ys ] }
    -- ArithSeqE Range                      -- { [ 1 ,2 .. 10 ] }
    wilderE (ListE es) = ListE (map wilderE es)
    wilderE (SigE e t) = SigE (wilderE e) t
    -- RecConE Name [FieldExp]              -- { T { x = y, z = w } }
    -- RecUpdE Exp [FieldExp]               -- { (f x) { z = w } }
    wilderE e = e

    wilderM :: Match -> Match
    wilderM (Match p b ds) = Match (wilder p) (wilderB b) (map wilderD ds)

    wilderD :: Dec -> Dec
    wilderD d = d -- error "djinn: no wilderD yet"

    wilderB :: Body -> Body
    wilderB b = b --error "djinn: no wilderD yet"

    {- |
    Generate an anonymous expression of the given type (if it is realizable).
    -}
    djinn :: Q Type -- ^ type
          -> Q Exp
    djinn qtyp = do
      typ <- qtyp
      djinn0 False Nothing typ

    {- |
    Generate a list of anonymous expressions of the given type (if it is realizable)↙
        ↳ .
    -}
    djinns :: Q Type -- ^ type
           -> Q Exp
    djinns qtyp = do
      typ <- qtyp
      djinn0 True Nothing typ

    {- |
    Generate a named declaration with an accompanying type signature.  For example:

    >   $(djinnD "maybeToEither" [t| forall a b . a ->  Maybe b ->  Either a b |])
    >   main = print . map (maybeToEither "foo") $ [ Nothing, Just "bar" ]

    might print @[Left \"foo\",Right \"bar\"]@.
    -}
    djinnD :: String  -- ^ name
```

```
        -> Q Type  -- ^ type
        -> Q [Dec]
djinnD str qtyp = do
  let name = mkName str
  typ <- qtyp
  exp' <- djinn0 False (Just str) typ
  return
    [ SigD name typ
    , FunD name [ Clause [] (NormalB $ exp') [] ] ]

{- |
Generate a named declaration with an accompanying type signature
for a list of possible realizations of a type.

>   $(djinnsD "picks" [t| forall a . (a, a) -> (a -> a) -> a |])
>   main = print [ p ("A","B") (++"C") | p <- picks ]

might print @["BC","AC","B","A"]@.

-}
djinnsD :: String  -- ^ name
        -> Q Type  -- ^ type
        -> Q [Dec]
djinnsD str qtyp = do
  let name = mkName str
  typ <- qtyp
  exp' <- djinn0 True (Just str) typ
  let ForallT vs cxt t = typ
  return
    [ SigD name (ForallT vs cxt (AppT ListT t))
    , FunD name [ Clause [] (NormalB $ exp') [] ] ]
```

6 src/Language/Haskell/Djinn/HTypes.hs

```
--
-- Modified to use TemplateHaskell by Claude Heiland-Allen, August 2010
--
-- Copyright (c) 2005 Lennart Augustsson
-- See LICENSE for licensing details.
--
module Language.Haskell.Djinn.HTypes(HKind(..), HType(..), HSymbol, ↙
    ↳ HEnvironment1, HEnvironment, hTypeToFormula,
        isHTUnion, getHTVars, substHT,
        HClause(..), HPat(..), HExpr(..), termToHExpr, termToHClause, ↙
            ↳ getBinderVars) where
import Language.Haskell.TH (Name, mkName)

import Data.List(union, (\\))
import Control.Monad(zipWithM)
import Language.Haskell.Djinn.LJTFormula (Formula(..), Term(..), ConsDesc(..), ↙
    ↳ Symbol(..))

type HSymbol = Name

data HKind
    = KStar
    | KArrow HKind HKind
```

```
    | KVar Int
    deriving (Eq, Show)

data HType
        = HTApp HType HType
        | HTVar HSymbol
        | HTCon HSymbol
        | HTTuple [HType]
        | HTArrow HType HType
        | HTUnion [(HSymbol, [HType])]          -- Only for data types; only at ↙
            ↳ top level
        deriving (Eq, Show)

type HEnvironment1 = (HSymbol, ([HSymbol], HType))
type HEnvironment  = [HEnvironment1]

isHTUnion :: HType -> Bool
isHTUnion (HTUnion _) = True
isHTUnion _ = False

{-
htNot :: HSymbol -> HType
htNot x = HTArrow (HTVar x) (HTCon "Void")
-}

getHTVars :: HType -> [HSymbol]
getHTVars (HTApp f a) = getHTVars f `union` getHTVars a
getHTVars (HTVar v) = [v]
getHTVars (HTCon _) = []
getHTVars (HTTuple ts) = foldr union [] (map getHTVars ts)
getHTVars (HTArrow f a) = getHTVars f `union` getHTVars a
getHTVars _ = error "getHTVars"

-------------------------------

hTypeToFormula :: HEnvironment -> HType -> Formula
hTypeToFormula ss (HTTuple ts) = Conj (map (hTypeToFormula ss) ts)
hTypeToFormula ss (HTArrow t1 t2) = hTypeToFormula ss t1 :-> hTypeToFormula ss ↙
    ↳ t2
hTypeToFormula ss (HTUnion ctss) = Disj [ (ConsDesc c (length ts), ↙
    ↳ hTypeToFormula ss (HTTuple ts)) | (c, ts) <- ctss ]
hTypeToFormula ss t =
    case expandSyn ss t [] of
    Nothing -> PVar $ SymbolS $ show t
    Just t' -> hTypeToFormula ss t'

expandSyn :: HEnvironment -> HType -> [HType] -> Maybe HType
expandSyn ss (HTApp f a) as = expandSyn ss f (a:as)
expandSyn ss (HTCon c) as =
    case lookup c ss of
    Just (vs, t) | length vs == length as -> Just $ substHT (zip vs as) t
    _ -> Nothing
expandSyn _ _ _ = Nothing

substHT :: [(HSymbol, HType)] -> HType -> HType
substHT r (HTApp f a) = HTApp (substHT r f) (substHT r a)
substHT r t@(HTVar v) =
```

```
75      case lookup v r of
        Nothing -> t
        Just t' -> t'
    substHT _ t@(HTCon _) = t
    substHT r (HTTuple ts) = HTTuple (map (substHT r) ts)
80  substHT r (HTArrow f a) = HTArrow (substHT r f) (substHT r a)
    substHT r (HTUnion (ctss)) = HTUnion [ (c, map (substHT r) ts) | (c, ts) <- ctss↙
        ↳ ]


    ------------------------------

85

    data HClause = HClause HSymbol [HPat] HExpr
        deriving (Show, Eq)

90  data HPat = HPVar HSymbol | HPCon HSymbol | HPTuple [HPat] | HPAt HSymbol HPat |↙
        ↳ HPApply HPat HPat
        deriving (Show, Eq)

    data HExpr = HELam [HPat] HExpr | HEApply HExpr HExpr | HECon HSymbol | HEVar ↙
        ↳ HSymbol | HETuple [HExpr] |
            HECase HExpr [(HPat, HExpr)]
95      deriving (Show, Eq)

    unSymbol :: Symbol -> HSymbol
    unSymbol (Symbol  s) =        s
    unSymbol (SymbolS s) = mkName s
100
    termToHExpr :: Term -> HExpr
    termToHExpr term = niceNames $ etaReduce $ remUnusedVars $ fst $ conv [] term
      where conv _vs (Var s) = (HEVar $ unSymbol s, [])
            conv vs (Lam s te) =
105             let hs = unSymbol s
                    (te', ss) = conv (hs : vs) te
                in  (hELam [convV hs ss] te', ss)
            conv vs (Apply (Cinj (ConsDesc s n) _) a) = (f $ foldl HEApply (HECon s)↙
                ↳ as, ss)
                where (f, as) = unTuple n ha
110                       (ha, ss) = conv vs a
            conv vs (Apply te1 te2) = convAp vs te1 [te2]
    --        conv _vs (Ctuple 0) = (HECon "()", [])
            conv _vs (Ctuple 0) = (HETuple [], [])
            conv _vs e = error $ "termToHExpr " ++ show e
115
            unTuple 0 _ = (id, [])
            unTuple 1 a = (id, [a])
            unTuple n (HETuple as) | length as == n = (id, as)
            unTuple n e = error $ "unTuple: unimplemented " ++ show (n, e)
120
            unTupleP 0 _ = []
    --      unTupleP 1 p = [p]
            unTupleP n (HPTuple ps) | length ps == n = ps
            unTupleP n p = error $ "unTupleP: unimplemented " ++ show (n, p)
125
            convAp vs (Apply te1 te2) as = convAp vs te1 (te2:as)
            convAp vs (Ctuple n) as | length as == n =
```

```
                let (es, sss) = unzip $ map (conv vs) as
                in  (hETuple es, concat sss)
        convAp vs (Ccases cds) (se : es) =
                let (alts, ass) = unzip $ zipWith cAlt es cds
                    cAlt (Lam v e) (ConsDesc c n) =
                        let hv = unSymbol v
                            (he, ss) = conv (hv : vs) e
                            ps = case lookup hv ss of
                                 Nothing -> replicate n underscore
                                 Just p -> unTupleP n p
                        in  ((foldl HPApply (HPCon c) ps, he), ss)
                    cAlt e _ = error $ "cAlt " ++ show e
                    (e', ess) = conv vs se
                in  (hECase e' alts, ess ++ concat ass)
        convAp vs (Csplit n) (b : a : as) =
                let (hb, sb) = conv vs b
                    (a', sa) = conv vs a
                    (as', sss) = unzip $ map (conv vs) as
                    (ps, b') = unLam n hb
                    unLam 0 e = ([], e)
                    unLam k (HELam ps0 e) | length ps0 >= n = let (ps1, ps2) = splitAt k ps0 in (ps1, hELam ps2 e)
                    unLam k e = error $ "unLam: unimplemented" ++ show (k, e)
                in  case a' of
                        HEVar v | v `elem` vs && null as -> (b', [(v, HPTuple ps)] ++ sb ++ sa)
                        _ -> (foldr HEApply (hECase a' [(HPTuple ps, b')]) as',
                              sb ++ sa ++ concat sss)

        convAp vs f as =
                let (es, sss) = unzip $ map (conv vs) (f:as)
                in  (foldl1 HEApply es, concat sss)

        convV hs ss =
                case lookup hs ss of
                Nothing -> HPVar hs
                Just p -> HPAt hs p

        hETuple [e] = e
        hETuple es = HETuple es

niceNames :: HExpr -> HExpr
niceNames e =
    let bvars = filter (/= mkName "_") $ getBinderVarsHE e
        chars = ['a'..'z']
        nvars = map (:[]) chars ++ [ cs ++ [c] | cs <- nvars, c <- chars ]
        freevars = getAllVars e \\ bvars
        vars = map mkName nvars \\ freevars
        sub = zip bvars vars
    in  hESubst sub e

hELam :: [HPat] -> HExpr -> HExpr
hELam [] e = e
hELam ps (HELam ps' e) = HELam (ps ++ ps') e
hELam ps e = HELam ps e

hECase :: HExpr -> [(HPat, HExpr)] -> HExpr
```

```
--hECase e [] = HEApply (HEVar "void") e
--hECase _ [(HPCon "()", e)] = e
hECase e pes | all (uncurry eqPatExpr) pes = e
hECase e [(p, HELam ps b)] = HELam ps $ hECase e [(p, b)]
hECase se alts@((_, HELam ops _):_) | m > 0 = HELam (take m ops) $ hECase se alts'
  where m = minimum (map (numBind . snd) alts)
        numBind (HELam ps _) = length (takeWhile isPVar ps)
        numBind _ = 0
        isPVar (HPVar _) = True
        isPVar _ = False
        alts' = [ let (ps1, ps2) = splitAt m ps in (cps, hELam ps2 $ hESubst (zipWith (\ (HPVar v) n -> (v, n)) ps1 ns) e)
                | (cps, HELam ps e) <- alts ]
        ns = [ n | HPVar n <- take m ops ]
-- if all arms are equal and there are at least two alternatives there can be no bound vars
-- from the patterns
hECase _ ((_,e):alts@(_:_)) | all (alphaEq e . snd) alts = e
hECase e alts = HECase e alts

eqPatExpr :: HPat -> HExpr -> Bool
eqPatExpr (HPVar s) (HEVar s') = s == s'
eqPatExpr (HPCon s) (HECon s') = s == s'
eqPatExpr (HPTuple ps) (HETuple es) = and (zipWith eqPatExpr ps es)
eqPatExpr (HPApply pf pa) (HEApply ef ea) = eqPatExpr pf ef && eqPatExpr pa ea
eqPatExpr _ _ = False

alphaEq :: HExpr -> HExpr -> Bool
alphaEq e1 e2 | e1 == e2 = True
alphaEq (HELam ps1 e1) (HELam ps2 e2) =
    Nothing /= do
        s <- matchPat (HPTuple ps1) (HPTuple ps2)
        if alphaEq (hESubst s e1) e2 then
            return ()
         else
            Nothing
alphaEq (HEApply f1 a1) (HEApply f2 a2) = alphaEq f1 f2 && alphaEq a1 a2
alphaEq (HECon s1) (HECon s2) = s1 == s2
alphaEq (HEVar s1) (HEVar s2) = s1 == s2
alphaEq (HETuple es1) (HETuple es2) | length es1 == length es2 = and (zipWith alphaEq es1 es2)
alphaEq (HECase e1 alts1) (HECase e2 alts2) =
    alphaEq e1 e2 && and (zipWith alphaEq [ HELam [p] e | (p, e) <- alts1 ] [ HELam [p] e | (p, e) <- alts2 ])
alphaEq _ _ = False

matchPat :: HPat -> HPat -> Maybe [(HSymbol, HSymbol)]
matchPat (HPVar s1) (HPVar s2) = return [(s1, s2)]
matchPat (HPCon s1) (HPCon s2) | s1 == s2 = return []
matchPat (HPTuple ps1) (HPTuple ps2) | length ps1 == length ps2 = do
    ss <- zipWithM matchPat ps1 ps2
    return $ concat ss
matchPat (HPAt s1 p1) (HPAt s2 p2) = do
    s <- matchPat p1 p2
    return $ (s1, s2) : s
matchPat (HPApply f1 a1) (HPApply f2 a2) = do
```

```
235         s1 <- matchPat f1 f2
            s2 <- matchPat a1 a2
            return $ s1 ++ s2
        matchPat _ _ = Nothing

240     hESubst :: [(HSymbol, HSymbol)] -> HExpr -> HExpr
        hESubst s (HELam ps e) = HELam (map (hPSubst s) ps) (hESubst s e)
        hESubst s (HEApply f a) = HEApply (hESubst s f) (hESubst s a)
        hESubst _ e@(HECon _) = e
        hESubst s (HEVar v) = HEVar $ maybe v id $ lookup v s
245     hESubst s (HETuple es) = HETuple (map (hESubst s) es)
        hESubst s (HECase e alts) = HECase (hESubst s e) [(hPSubst s p, hESubst s b) | (↙
            ↳ p, b) <- alts]

        hPSubst :: [(HSymbol, HSymbol)] -> HPat -> HPat
        hPSubst s (HPVar v) = HPVar $ maybe v id $ lookup v s
250     hPSubst _ p@(HPCon _) = p
        hPSubst s (HPTuple ps) = HPTuple (map (hPSubst s) ps)
        hPSubst s (HPAt v p) = HPAt (maybe v id $ lookup v s) (hPSubst s p)
        hPSubst s (HPApply f a) = HPApply (hPSubst s f) (hPSubst s a)

255

        termToHClause :: HSymbol -> Term -> HClause
        termToHClause i term =
            case termToHExpr term of
            HELam ps e -> HClause i ps e
260         e -> HClause i [] e

        remUnusedVars :: HExpr -> HExpr
        remUnusedVars expr = fst $ remE expr
          where remE (HELam ps e) =
265                   let (e', vs) = remE e
                      in  (HELam (map (remP vs) ps) e', vs)
                remE (HEApply f a) =
                      let (f', fs) = remE f
                          (a', as) = remE a
270                   in  (HEApply f' a', fs ++ as)
                remE (HETuple es) =
                      let (es', sss) = unzip (map remE es)
                      in  (HETuple es', concat sss)
                remE (HECase e alts) =
275                   let (e', es) = remE e
                          (alts', sss) = unzip [ let (ee', ss) = remE ee in ((remP ss p, ↙
                              ↳ ee'), ss) | (p, ee) <- alts ]
                      in  case alts' of
                          [(u, b)] | u == underscore -> (b, concat sss)
                          _ -> (hECase e' alts', es ++ concat sss)
280         remE e@(HECon _) = (e, [])
            remE e@(HEVar v) = (e, [v])
            remP vs p@(HPVar v) = if v `elem` vs then p else underscore
            remP _vs p@(HPCon _) = p
            remP vs (HPTuple ps) = hPTuple (map (remP vs) ps)
285         remP vs (HPAt v p) = if v `elem` vs then HPAt v (remP vs p) else remP vs↙
                ↳ p
            remP vs (HPApply f a) = HPApply (remP vs f) (remP vs a)
            hPTuple ps | all (== underscore) ps = underscore
            hPTuple ps = HPTuple ps
```

```
290 underscore :: HPat
    underscore = HPVar (mkName "_")

    getBinderVars :: HClause -> [HSymbol]
    getBinderVars (HClause _ pats expr) = concatMap getBinderVarsHP pats ++ ↙
        ↳ getBinderVarsHE expr
295
    getBinderVarsHE :: HExpr -> [HSymbol]
    getBinderVarsHE expr = gbExp expr
      where gbExp (HELam ps e) = concatMap getBinderVarsHP ps ++ gbExp e
            gbExp (HEApply f a) = gbExp f ++ gbExp a
300         gbExp (HETuple es) = concatMap gbExp es
            gbExp (HECase se alts) = gbExp se ++ concatMap (\ (p, e) -> ↙
                ↳ getBinderVarsHP p ++ gbExp e) alts
            gbExp _ = []

    getBinderVarsHP :: HPat -> [HSymbol]
305 getBinderVarsHP pat = gbPat pat
      where gbPat (HPVar s) = [s]
            gbPat (HPCon _) = []
            gbPat (HPTuple ps) = concatMap gbPat ps
            gbPat (HPAt s p) = s : gbPat p
310         gbPat (HPApply f a) = gbPat f ++ gbPat a

    getAllVars :: HExpr -> [HSymbol]
    getAllVars expr = gaExp expr
      where gaExp (HELam _ps e) = gaExp e
315         gaExp (HEApply f a) = gaExp f `union` gaExp a
            gaExp (HETuple es) = foldr union [] (map gaExp es)
            gaExp (HECase se alts) = foldr union (gaExp se) (map (\ (_p, e) -> gaExp↙
                ↳  e) alts)
            gaExp (HEVar s) = [s]
            gaExp _ = []
320
    etaReduce :: HExpr -> HExpr
    etaReduce expr = fst $ eta expr
      where eta (HELam [HPVar v] (HEApply f (HEVar v'))) | v == v' && v `notElem` vs↙
          ↳  = (f', vs)
                where (f', vs) = eta f
325         eta (HELam ps e) = (HELam ps e', vs) where (e', vs) = eta e
            eta (HEApply f a) = (HEApply f' a', fvs++avs) where (f', fvs) = eta f; (↙
                ↳ a', avs) = eta a
            eta e@(HECon _) = (e, [])
            eta e@(HEVar s) = (e, [s])
            eta (HETuple es) = (HETuple es', concat vss) where (es', vss) = unzip $ ↙
                ↳ map eta es
330         eta (HECase e alts) = (HECase e' alts', vs ++ concat vss) where (e', vs)↙
                ↳  = eta e
                                                                    (alts', ↙
                                                                        ↳ vss↙
                                                                        ↳ ) ↙
                                                                        ↳ = ↙
                                                                        ↳ unzip↙
                                                                        ↳  $↙
                                                                        ↳  [↙
                                                                        ↳  ↙
```

```
    ↳ let↙
    ↳  (↙
    ↳ a↙
    ↳ ',↙
    ↳  ↙
    ↳ ss↙
    ↳ ) ↙
    ↳ = ↙
    ↳ eta↙
    ↳  a↙
    ↳  ↙
    ↳ in↙
    ↳  ↙
    ↳ ((↙
    ↳ p,↙
    ↳  a↙
    ↳ ')↙
    ↳ , ↙
    ↳ ss↙
    ↳ )
```

7 src/Language/Haskell/Djinn/LJTFormula.hs

```
--
-- Modified to use TemplateHaskell by Claude Heiland-Allen, August 2010
--
-- Copyright (c) 2005 Lennart Augustsson
-- See LICENSE for licensing details.
--
module Language.Haskell.Djinn.LJTFormula(Symbol(..), Formula(..), (<->), (&), {-↙
    ↳  (|:), -} fnot, false, true,
         ConsDesc(..),
         Term(..), applys, freeVars
         ) where
import Data.List(union, (\\))
import Language.Haskell.TH (Name)

infixr 2 :->
infix  2 <->
--infixl 3 |:
infixl 4 &
```

```
data Symbol = Symbol Name | SymbolS String
      deriving (Eq, Ord, Show)

data ConsDesc = ConsDesc Name Int      -- name and arity
      deriving (Eq, Ord, Show)

data Formula
        = Conj [Formula]
        | Disj [(ConsDesc, Formula)]
        | Formula :-> Formula
        | PVar Symbol
      deriving (Eq, Ord, Show)

(<->) :: Formula -> Formula -> Formula
x <-> y = (x:->y) & (y:->x)

(&) :: Formula -> Formula -> Formula
x & y = Conj [x, y]

{-
(|:) :: Formula -> Formula -> Formula
x |: y = Disj [((ConsDesc "Left" 1), x), ((ConsDesc "Right" 1), y)]
-}

fnot :: Formula -> Formula
fnot x = x :-> false

false :: Formula
false = Disj []

true :: Formula
true = Conj []

------------------------------

data Term
        = Var Symbol
        | Lam Symbol Term
        | Apply Term Term
        | Ctuple Int
        | Csplit Int
        | Cinj ConsDesc Int
        | Ccases [ConsDesc]
        | Xsel Int Int Term              --- XXX just temporary by MJ
    deriving (Eq, Ord, Show)

applys :: Term -> [Term] -> Term
applys f as = foldl Apply f as

freeVars :: Term -> [Symbol]
freeVars (Var s) = [s]
freeVars (Lam s e) = freeVars e \\ [s]
freeVars (Apply f a) = freeVars f `union` freeVars a
freeVars (Xsel _ _ e) = freeVars e
freeVars _ = []
```

8 src/Language/Haskell/Djinn/LJT.hs

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
--
-- Modified to use Template Haskell by Claude Heiland-Allen, August 2010
--
-- Copyright (c) 2005, 2008 Lennart Augustsson
-- See LICENSE for licensing details.
--
-- Intuitionistic theorem prover
-- Written by Roy Dyckhoff, Summer 1991
-- Modified to use the LWB syntax  Summer 1997
-- and simplified in various ways...
--
-- Translated to Haskell by Lennart Augustsson December 2005
--
-- Incorporates the Vorob'ev-Hudelmaier etc calculus (I call it LJT)
-- See RD's paper in JSL 1992:
-- "Contraction-free calculi for intuitionistic logic"
--
-- Torkel Franzen (at SICS) gave me good ideas about how to write this
-- properly, taking account of first-argument indexing,
-- and I learnt a trick or two from Neil Tennant's "Autologic" book.

module Language.Haskell.Djinn.LJT (
            module Language.Haskell.Djinn.LJTFormula, provable,
            prove, Proof) where

import Language.Haskell.TH (newName, Q)

import Control.Monad (liftM, liftM2, foldM)
import Control.Monad.Logic (
  LogicT, msplit, observeAllT, MonadLogic, MonadTrans(..), MonadPlus(..))

import Data.List (partition)
import Debug.Trace (trace)

import Language.Haskell.Djinn.LJTFormula (
  Symbol(..), Formula(..), Term(..), ConsDesc(..), false, applys)

mtrace :: String -> a -> a
mtrace m x = if debug then trace m x else x
-- wrap :: (Show a, Show b) => String -> a -> b -> b
-- wrap fun args ret = mtrace (fun ++ ": " ++ show args) $
--                     let o = show ret in seq o $
--                     mtrace (fun ++ " returns: " ++ o) ret
wrapM :: (Show a, Show b, Monad m) => String -> a -> m b -> m b
wrapM fun args mret = do
    () <- mtrace (fun ++ ": " ++ show args) $ return ()
    ret <- mret
    () <- mtrace (fun ++ " returns: " ++ show ret) $ return ()
    return ret
debug :: Bool
debug = False

type MoreSolutions = Bool

```

```
provable :: Formula -> Q Bool
provable a = null `fmap` prove False [] a

prove :: MoreSolutions -> [(Symbol, Formula)] -> Formula -> Q [Proof]
prove more env a = runP $ redtop more env a

redtop :: MoreSolutions -> [(Symbol, Formula)] -> Formula -> P Proof
redtop more ifs a = do
    let form = foldr (:->) a (map snd ifs)
    p <- redant more [] [] [] [] form
    nf (foldl Apply p (map (Var . fst) ifs))

------------------------------
-----
type Proof = Term

subst :: Term -> Symbol -> Term -> P Term
subst b x term = sub term
  where sub t@(Var s') = if x == s' then copy [] b else return t
        sub (Lam s t) = liftM (Lam s) (sub t)
        sub (Apply t1 t2) = liftM2 Apply (sub t1) (sub t2)
        sub t = return t

copy :: [(Symbol, Symbol)] -> Term -> P Term
copy r (Var s) = return $ Var $ maybe s id $ lookup s r
copy r (Lam s t) = do
    s' <- newSym "c"
    liftM (Lam s') $ copy ((s, s'):r) t
copy r (Apply t1 t2) = liftM2 Apply (copy r t1) (copy r t2)
copy _r t = return t

------------------------------

applyAtom :: Term -> Term -> Term
applyAtom f a = Apply f a

curryt :: Int -> Term -> P Term
curryt n p = do
  xs <- mapM (\i -> newSym $ "x_" ++ show i) [0 .. n-1]
  return $ foldr Lam (Apply p (applys (Ctuple n) (map Var xs))) xs

inj :: ConsDesc -> Int -> Term -> P Term
inj cd i p = do
  x <- newSym "x"
  return $ Lam x $ Apply p (Apply (Cinj cd i) (Var x))

applyImp :: Term -> Term -> P Term
applyImp p q = do
  x <- newSym "x"
  y <- newSym "y"
  return $ Apply p (Apply q (Lam y $ Apply p (Lam x (Var y))))

-- ((c->d)->false) -> ((c->false)->false, d->false)
-- p : (c->d)->false)
-- replace p1 and p2 with the components of the pair
cImpDImpFalse :: Symbol -> Symbol -> Term -> Term -> P Term
cImpDImpFalse p1 p2 cdf gp = do
```

```
      [cf, x, d, c] <- mapM newSym ["cf", "x", "d", "c"]
      let p1b = Lam cf $ Apply cdf $ Lam x $ Apply (Ccases []) $ Apply (Var cf) (↙
          ↳ Var x)
          p2b = Lam d $ Apply cdf $ Lam c $ Var d
      subst p1b p1 gp >>= subst p2b p2

------------------------------

-- More simplifications:
--  split where no variables used can be removed
--  either with equal RHS can me merged.

-- Compute the normal form
nf :: Term -> P Term
nf ee = spine ee []
  where spine (Apply f a) as = do a' <- nf a; spine f (a' : as)
        spine (Lam s e) [] = liftM (Lam s) (nf e)
        spine (Lam s e) (a : as) = do e' <- subst a s e; spine e' as
        spine (Csplit n) (b : tup : args) | istup && n <= length xs = spine (↙
            ↳ applys b xs) args
          where (istup, xs) = getTup tup
                getTup (Ctuple _) = (True, [])
                getTup (Apply f a) = let (tf, as) = getTup f in (tf, a:as)
                getTup _ = (False, [])
        spine (Ccases []) (e@(Apply (Ccases []) _) : as) = spine e as
        spine (Ccases cds) (Apply (Cinj _ i) x : as) | length as >= n = spine (↙
            ↳ Apply (as!!i) x) (drop n as)
              where n = length cds
        spine f as = return $ applys f as


------------------------------
----- Our Proof monad, P, a monad transformer with multiple results

newtype PT q a = P{ unP :: LogicT q a } -- thanks kmc, Cale, #haskell
  deriving (Functor, Monad, MonadPlus, MonadLogic, MonadTrans)
type P a = PT Q a
liftQ :: Q a -> P a
liftQ = lift

none :: P a
none = mzero

many :: [a] -> P a
many = foldr (\x y -> return x `mplus` y) mzero

atMostOne :: P a -> P a
atMostOne m = do
  p <- msplit m
  case p of
    Nothing    -> mzero
    Just (a,_) -> return a

runP :: P a -> Q [a]
runP (P l) = observeAllT l

```

```
-------------------------------
----- Atomic formulae
data AtomF = AtomF Term Symbol
    deriving (Eq)
instance Show AtomF where
    show (AtomF p s) = show p ++ ":" ++ show s

type AtomFs = [AtomF]

findAtoms :: Symbol -> AtomFs -> [Term]
findAtoms s atoms = [ p | AtomF p s' <- atoms, s == s' ]

--removeAtom :: Symbol -> AtomFs -> AtomFs
--removeAtom s atoms = [ a | a@(AtomF _ s') <- atoms, s /= s' ]

addAtom :: AtomF -> AtomFs -> AtomFs
addAtom a as = if a `elem` as then as else a : as

-------------------------------
----- Implications of one atom

data AtomImp = AtomImp Symbol Antecedents
     deriving (Show)
type AtomImps = [AtomImp]

extract :: AtomImps -> Symbol -> ([Antecedent], AtomImps)
extract aatomImps@(atomImp@(AtomImp a' bs) : atomImps) a =
    case compare a a' of
    GT -> let (rbs, restImps) = extract atomImps a in (rbs, atomImp : restImps)
    EQ -> (bs, atomImps)
    LT -> ([], aatomImps)
extract _ _ = ([], [])

insert :: AtomImps -> AtomImp -> AtomImps
insert [] ai = [ ai ]
insert aatomImps@(atomImp@(AtomImp a' bs') : atomImps) ai@(AtomImp a bs) =
    case compare a a' of
    GT -> atomImp : insert atomImps ai
    EQ -> AtomImp a (bs ++ bs') : atomImps
    LT -> ai : aatomImps

-------------------------------
----- Nested implications, (a -> b) -> c

data NestImp = NestImp Term Formula Formula Formula -- NestImp a b c represents ↙
    ↳ (a :-> b) :-> c
    deriving (Eq)
instance Show NestImp where
    show (NestImp _ a b c) = show $ (a :-> b) :-> c

type NestImps = [NestImp]

addNestImp :: NestImp -> NestImps -> NestImps
addNestImp n ns = if n `elem` ns then ns else n : ns

-------------------------------
----- Ordering of nested implications
```

```
heuristics :: Bool
heuristics = True

order :: NestImps -> Formula -> AtomImps -> NestImps
order nestImps g atomImps =
    if heuristics then
        nestImps
    else
        let
            good_for (NestImp _ _ _ (Disj [])) = True
            good_for (NestImp _ _ _ g') = g == g'
            nice_for (NestImp _ _ _ (PVar s)) =
                case extract atomImps s of
                (bs', _) -> let bs = [ b | A _ b <- bs'] in g `elem` bs || false
                    ↳ `elem` bs
            nice_for _ = False
            (good, ok) = partition good_for nestImps
            (nice, bad) = partition nice_for ok
        in  good ++ nice ++ bad

-------------------------------
----- Generate a new unique variable
newSym :: String -> P Symbol
newSym s = Symbol `fmap` liftQ (newName s)

-------------------------------
----- Generate all ways to select one element of a list
select :: [a] -> P (a, [a])
select zs = many [ del n zs | n <- [0 .. length zs - 1] ]
  where del 0 (x:xs) = (x, xs)
        del n (x:xs) = let (y,ys) = del (n-1) xs in (y, x:ys)
        del _ _ = error "select"

-------------------------------
-----

data Antecedent = A Term Formula deriving (Show)
type Antecedents = [Antecedent]

type Goal = Formula

--
-- This is the main loop of the proof search.
--
-- The redant functions reduce antecedents and the redsucc
-- function reduces the goal (succedent).
--
-- The antecedents are kept in four groups: Antecedents, AtomImps, NestImps,
    ↳ AtomFs
--   Antecedents contains as yet unclassified antecedents; the redant functions
--     go through them one by one and reduces and classifies them.
--   AtomImps contains implications of the form (a -> b), where `a` is an atom.
--     To speed up the processing it is stored as a map from the `a` to all the
--     formulae it implies.
--   NestImps contains implications of the form ((b -> c) -> d)
--   AtomFs contains atomic formulae.
--
```

```
-- There is also a proof object associated with each antecedent.
--
redant :: MoreSolutions -> Antecedents -> AtomImps -> NestImps -> AtomFs -> Goal
    -> P Proof
redant more antes atomImps nestImps atoms goal =
    wrapM "redant" (antes, atomImps, nestImps, atoms, goal) $
    case antes of
    [] -> redsucc goal
    a:l -> redant1 a l goal
  where redant0 l g = redant more l atomImps nestImps atoms g
        redant1 :: Antecedent -> Antecedents -> Goal -> P Proof
        redant1 a@(A p f) l g =
            wrapM "redant1" ((a, l), atomImps, nestImps, atoms, g) $
            if f == g then
                -- The goal is the antecedent, we're done.
                -- XXX But we might want more?
                if more then
                    return p `mplus` redant1' a l g
                else
                    return p
            else
                redant1' a l g

        -- Reduce the first antecedent
        redant1' :: Antecedent -> Antecedents -> Goal -> P Proof
        redant1' (A p (PVar s)) l g =
           let af = AtomF p s
               (bs, restAtomImps) = extract atomImps s
           in  redant more ([A (Apply f p) b | A f b <- bs] ++ l) restAtomImps
               nestImps (addAtom af atoms) g
        redant1' (A p (Conj bs)) l g = do
           vs <- mapM (const (newSym "v")) bs
           gp <- redant0 (zipWith (\ v a -> A (Var v) a) vs bs ++ l) g
           return $ applys (Csplit (length bs)) [foldr Lam gp vs, p]
        redant1' (A p (Disj ds)) l g = do
           vs <- mapM (const (newSym "d")) ds
           ps <- mapM (\ (v, (_, d)) -> redant1 (A (Var v) d) l g) (zip vs ds)
           if null ds && g == Disj [] then
               -- We are about to construct `void p : Void', so we shortcut
               -- it with just `p'.
               return p
            else
               return $ applys (Ccases (map fst ds)) (p : zipWith Lam vs ps)
        redant1' (A p (a :-> b)) l g = redantimp p a b l g

        redantimp :: Term -> Formula -> Formula -> Antecedents -> Goal -> P
            Proof
        redantimp t c d a g =
            wrapM "redantimp" (c,d,a,g) $
            redantimp' t c d a g

        -- Reduce an implication antecedent
        redantimp' :: Term -> Formula -> Formula -> Antecedents -> Goal -> P
            Proof
        -- p : PVar s -> b
        redantimp' p (PVar s) b l g = redantimpatom p s b l g
        -- p : (c & d) -> b
```

```
redantimp' p (Conj cs) b l g = do
    x <- newSym "x"
    let imp = foldr (:->) b cs
    gp <- redant1 (A (Var x) imp) l g
    cry <- curryt (length cs) p
    subst cry x gp
-- p : (c | d) -> b
redantimp' p (Disj ds) b l g = do
    vs <- mapM (const (newSym "d")) ds
    gp <- redant0 (zipWith (\ v (_, d) -> A (Var v) (d :-> b)) vs ds ++ l) g
    foldM (\ r (i, v, (cd, _)) -> inj cd i p >>= \nj -> subst nj v r) gp (zip3 [0..] vs ds)
-- p : (c -> d) -> b
redantimp' p (c :-> d) b l g = redantimpimp p c d b l g

redantimpimp :: Term -> Formula -> Formula -> Formula -> Antecedents -> Goal -> P Proof
redantimpimp f b c d a g =
    wrapM "redantimpimp" (b,c,d,a,g) $
    redantimpimp' f b c d a g

-- Reduce a double implication antecedent
redantimpimp' :: Term -> Formula -> Formula -> Formula -> Antecedents -> Goal -> P Proof
-- next clause exploits ~(C->D) <=> (~~C & ~D)
-- which isn't helpful when D = false
redantimpimp' p c d (Disj []) l g | d /= false = do
    x <- newSym "x"
    y <- newSym "y"
    gp <- redantimpimp (Var x) c false false (A (Var y) (d :-> false) : l) g
    cImpDImpFalse x y p gp
-- p : (c -> d) -> b
redantimpimp' p c d b l g = redant more l atomImps (addNestImp (NestImp p c d b) nestImps) atoms g

-- Reduce an atomic implication
redantimpatom :: Term -> Symbol -> Formula -> Antecedents -> Goal -> P Proof
redantimpatom p s b l g =
    wrapM "redantimpatom" (s,b,l,g) $
    redantimpatom' p s b l g

redantimpatom' :: Term -> Symbol -> Formula -> Antecedents -> Goal -> P Proof
redantimpatom' p s b l g =
  do
    a <- cutSearch more $ many (findAtoms s atoms)
    x <- newSym "x"
    gp <- redant1 (A (Var x) b) l g
    mtrace "redantimpatom: LLL" $
     subst (applyAtom p a) x gp
  `mplus`
    (mtrace "redantimpatom: RRR" $
     redant more l (insert atomImps (AtomImp s [A p b])) nestImps atoms g)
```

```
    {-
380             let ps = wrap "redantimpatom findAtoms" atoms $ findAtoms s atoms
                in  if not (null ps) then do
                        a <- cutSearch more $ many ps
                        x <- newSym "x"
                        gp <- redant1 (A (Var x) b) l g
385                     mtrace "redantimpatom: LLL" $
                         subst (applyAtom p a) x gp
                    else
                        mtrace "redantimpatom: RRR" $
                         redant more l (insert atomImps (AtomImp s [A p b])) ↙
                             ↳ nestImps atoms g
390 -}
        -- Reduce the goal, with all antecedents already being classified
        redsucc :: Goal -> P Proof
        redsucc g =
            wrapM "redsucc" (g, atomImps, nestImps, atoms) $
395         redsucc' g

        redsucc' :: Goal -> P Proof
        redsucc' a@(PVar s) =
            (cutSearch more $ many (findAtoms s atoms))
400       `mplus`
            -- The posin check is an optimization.  It gets a little slower ↙
                ↳ without the test.
            (if posin s atomImps nestImps then
                redsucc_choice a
            else
405             none)
        redsucc' (Conj cs) = do
            ps <- mapM redsucc cs
            return $ applys (Ctuple (length cs)) ps
        -- next clause deals with succedent (A v B) by pushing the
410     -- non-determinism into the treatment of implication on the left
        redsucc' (Disj ds) = do
            s1 <- newSym "_"
            let v = PVar s1
            redant0 [ A (Cinj cd i) $ d :-> v | (i, (cd, d)) <- zip [0..] ds ] v
415     redsucc' (a :-> b) = do
            s <- newSym "x"
            p <- redant1 (A (Var s) a) [] b
            return $ Lam s p

420     -- Now we have the hard part; maybe lots of formulae
        -- of form (C->D)->B  in nestImps to choose from!
        -- Which one to take first? We user the order heuristic.
        redsucc_choice :: Goal -> P Proof
        redsucc_choice g =
425         wrapM "redsucc_choice" g $
            redsucc_choice' g

        redsucc_choice' :: Goal -> P Proof
        redsucc_choice' g = do
430         let ordImps = order nestImps g atomImps
            (NestImp p c d b, restImps) <-
                mtrace ("redsucc_choice: order=" ++ show ordImps) $
                select ordImps
```

```
            x <- newSym "x"
            z <- newSym "z"
            qz <- redant more [A (Var z) $ d :-> b] atomImps restImps atoms (c ↙
                ↳ :-> d)
            gp <- redant more [A (Var x) b] atomImps restImps atoms g
            ai <- applyImp p (Lam z qz)
            subst ai x gp

posin :: Symbol -> AtomImps -> NestImps -> Bool
posin g atomImps nestImps = posin1 g atomImps || posin2 g [ (a :-> b) :-> c | ↙
    ↳ NestImp _ a b c <- nestImps ]

posin1 :: Symbol -> AtomImps -> Bool
posin1 g atomImps = any (\ (AtomImp _ bs) -> posin2 g [ b | A _ b <- bs]) ↙
    ↳ atomImps

posin2 :: Symbol -> [Formula] -> Bool
posin2 g bs = any (posin3 g) bs

posin3 :: Symbol -> Formula -> Bool
posin3 g (Disj as) = all (posin3 g) (map snd as)
posin3 g (Conj as) = any (posin3 g) as
posin3 g (_ :-> b) = posin3 g b
posin3 s (PVar s') = s == s'

cutSearch :: MoreSolutions -> P a -> P a
cutSearch False p = atMostOne p
cutSearch True p = p

--------------------------
```

9 TODO

```
support contexts / classes / methods
```