CS 4Z03 - Functional Programming, in Application to Interactive Web Interfaces for Discrete Mathematics Education

Peter Santos

August 9, 2010

This is a modified tautology checker that checks if two propositional statements are equal in their evaluation. It comes from the Tautology checker example in section 10.4 of *Programming in Haskell*, by Graham Hutton. This propositional checker no longer evaluates if a proposition a tautology, though two extra lines of code could enable this feature.

To begin, **Data.Set** is imported to allow for a more intuitive way of manipulating the lists that will be used in determining whether a proposition is equivalent to another, which will be discussed later.

```
module PropChecker where
import qualified Data.Set as Set
import Data.Set (Set)
```

A new data type is now defined which represents common propositional logic connectives such as *Or*, *And*, *Implies*, etc.

```
type Var = Char
data Prop = Const Bool
          | Var Var
          | Not Prop
          | And Prop Prop
          | Or Prop Prop
          | Imply Prop Prop
          | Equiv Prop Prop
          | Equal Prop Prop
        deriving (Show, Eq)
```

Subst will act kind of like a substitution since it doesn't really substitute variables, but rather is type that identifies what Bools should be used for what variables.

```
type Subst = Assoc Char Bool
```

Assoc acts lookup table for variables to bools, although this a general definition which only requires two different types, that may or may not be Char and Bool.

```
type Assoc k v = [(k, v)]
type Rel k v = Set (k, v)
  -- type Fct k v = Map.Map k v
```

find will take a variable(k) and table of variables to bools and find the first bool value for k and return it.

```
find     :: Eq k ⇒ k → Assoc k v → v
find k t = head [v | (k′, v) ← t, k ≡ k′]
```

Now that we have defined a way to associate bool values to variables, we can now use this in another function that recursively evaluates a proposition of type Prop. Using the find function it will find the appropriate bool value for a variable in a proposition and then evaluate the proposition once it has been constructed.

```
eval                :: Subst → Prop → Bool
eval _ (Const b)    = b
eval s (Var x)      = find x s
eval s (Not p)      = ¬ (eval s p)
eval s (And p q)    = eval s p ∧ eval s q
eval s (Or p q)     = eval s p ∨ eval s q
eval s (Imply p q)  = eval s p ⩽ eval s q
eval s (Equiv p q)  = eval s (Imply p q) ∧ eval s (Imply q p)
eval s (Equal p q)  = eval s p ≡ eval s q
```

We will use 'vars' to construct a list of variables, that are in a proposition. This function will produce duplicate variables, but that's okay for now, because we will remove them later.

```
vars              :: Prop → [Char]
vars (Const _)    = []
vars (Var x)      = [x]
vars (Not p)      = vars p
vars (And p q)    = vars p ++ vars q
vars (Or p q)     = vars p ++ vars q
vars (Imply p q)  = vars p ++ vars q
vars (Equiv p q)  = vars p ++ vars q
vars (Equal p q)  = vars p ++ vars q
```

The 'bools' function creates a complete truth table of all possible True and False combinations for a given number of variables.

```
bools         :: Int → [[Bool]]
bools 0       = [[]]
bools (n + 1) = map (False:) bss ++ map (True:) bss
                where bss = bools n
```

This function filters out all the duplicates of a list, and is used to remove the duplicate variables from the list that is generated from 'vars'.

```
rmdups        :: Eq a ⇒ [a] → [a]
rmdups []     = []
rmdups (x : xs) = x : rmdups (filter (≢ x) xs)
```

The function 'substs' will pair the variables produced from 'vars' with the list of bools generated from 'bools'.

```
substs   :: Prop → [Subst]
substs p = map (zip vs) (bools (length vs))
           where vs = rmdups (vars p)
```

```
type Rests                 = [Prop]
  -- This function will filter out all the props that don't satisfy the given
  -- restrictions
cleanSubst                 :: [Subst] → Prop → [Subst]
cleanSubst [] p            = []
cleanSubst (s : subs) p    = if eval s p then
                               s : cleanSubst subs p
                             else
                               cleanSubst subs p
  -- eval s p = flip eval p s = s ‘eval‘ p = (‘eval‘p) s = (λs → eval s p) s
  -- flip cleanSubst p = filter (flip eval p)
  -- This will purge the substition list of any substitutions that do not satisfy
  -- the given restrictions. ie, A /= B
```

```
readySubst                  :: [Subst] → Rests → [Subst]
readySubst subs []          = subs
readySubst subs (p : ps)    = readySubst (cleanSubst subs p) ps
  -- This will create the one part of the final substitution list
keepSubst :: [Subst] → Set.Set (Char, Bool) → [(Subst, Subst)]
keepSubst subs set1
              = [(s, Set.toList set1) | s ← subs, set1 `Set.isSubsetOf` Set.fromList s]
  -- This will create the final substitution list. The first arguments should
  -- take a larger clean substitution list and a smaller substitution list
  -- if the last substitution list is of equal size, then isEquiv should be
  -- used instead
finalSubst                  :: [Subst] → [Subst] → [(Subst, Subst)]
finalSubst subs1 subs2
                = foldl (++) [] [keepSubst subs1 (Set.fromList s) | s ← subs2]
  -- This function works with two propositions that have the same amount of
  -- variables. Boring, I know.
isEquiv                     :: Prop → Prop → Rests → Bool
isEquiv p1 p2 r             = if p1 ≡ p2 then
                                  True
                                else
                                  and [eval s p1 ≡ eval s p2 | s ← subs]
                                    where subs = readySubst (substs p1) r
  -- This function works with two propositions that have an unequal amount of
  -- variables. The substitution list always has the propositions with more
  -- variables p1.
isEquiv'                    :: Prop → Prop → [(Subst, Subst)] → Bool
isEquiv' p1 p2 subs         = and [eval s1 p1 ≡ eval s2 p2 | (s1, s2) ← subs]
  -- Give propMachine two propositions and it's restrictions and it will tell you
  -- If they are equivalent
propMachine                 :: Prop → Prop → Rests → Bool
propMachine p1 p2 r         = if varLp1 > varLp2 then
                                  isEquiv' p1 p2 (finalSubst (cSub1) (substs p2))
                                else if varLp2 > varLp1 then
                                  isEquiv' p2 p1 (finalSubst (cSub2) (substs p1))
                                else
                                  isEquiv p1 p2 r
                                  where cSub1 = readySubst (substs p1) r
                                    cSub2 = readySubst (substs p2) r
                                    varLp1 = length (rmdups (vars p1))
                                    varLp2 = length (rmdups (vars p2))
  -- This will give the first instance of when two propositions disagree
  -- and format it into a string using disagreeM.
disagree            :: Prop → Prop → Rests → String
disagree p1 p2 r    = if varLp1 > varLp2 then
                              disagreeM (fstFalsePair p1 p2 (finalSubst (cSub1) (substs p2)))
                        else if varLp2 > varLp1 then
                              disagreeM (fstFalsePair p2 p1 (finalSubst (cSub2) (substs p1)))
                        else
                              disagreeM (fstFalsePair p1 p2 (finalSubst cSub1 cSub2))
                              where cSub1 = readySubst (substs p1) r
                                cSub2 = readySubst (substs p2) r
                                varLp1 = length (rmdups (vars p1))
                                varLp2 = length (rmdups (vars p2))
  -- Takes a substitution and translates to a string.
disagreeM           :: Maybe Subst → String
```

```
disagreeM (Just s)  = disagreeM' s
disagreeM Nothing  = "your proposition is evaluated, it violates the given " ++
                     "restrictions. "
disagreeM'              :: Subst → String
disagreeM' ((v, b) : []) =    v : " = " ++ (show b) ++ ", your proposition doesn't " ++
                              "correctly describe the situation. "
disagreeM' ((v, b) : subs) = v : " = " ++ (show b) ++ ", " ++ disagreeM' subs
fstFalse                :: Maybe (Subst, Subst) → Maybe Subst
fstFalse subPair        =    case subPair of
                               Just (s1, s2) → Just (rmdups (s1 ++ s2))
                               Nothing → Nothing
fstFalsePair          :: Prop → Prop → [(Subst, Subst)] → Maybe Subst
fstFalsePair p1 p2 s = case (safeHead (takeWhile (λ(s1, s2) → eval s1 p1 ≢ eval s2 p2) s)) of
                          Just subPair → fstFalse (Just subPair)
                          Nothing → fstFalse Nothing
safeHead (h: _) = Just h
safeHead _ = Nothing

  -- Test propositions
  --
p1 :: Prop
p1 =  And (Var 'A') (Not (Var 'A'))
p2 :: Prop
p2 =  Imply (And (Var 'A') (Var 'B')) (Var 'A')
p3 :: Prop
p3 =  Imply (Var 'A') (And (Var 'A') (Var 'B'))
p4 :: Prop
p4 =  Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (Var 'B')
p5 :: Prop
p5 =  Or (Not (Var 'A')) (Var 'A')
p6 :: Prop
p6 =  Equiv (Imply (Var 'A') (Var 'B')) (Imply (Not (Var 'B')) (Not (Var 'A')))
p7 :: Prop
p7 =  Equiv (Not (And (Var 'A') (Var 'B'))) (Or (Not (Var 'A')) (Not (Var 'B')))
p8 :: Prop
p8 =  Imply (Var 'A') (Or (Var 'B') (Var 'C'))
p9 :: Prop
p9 =  Or (Imply (Var 'A') (Var 'B')) (Imply (Var 'A') (Var 'C'))
  -- Ladies or Tigers example
p10 :: Prop
p10 = Equiv (Var 'a') (Var 'd')
p11 :: Prop
p11 = And (Or (Var 'a') (Var 'b')) (Or (Var 'c') (Var 'd'))
r1 :: Prop
r1 = Not (Equal (Var 'a') (Var 'c'))
r2 :: Prop
r2 = Not (Equal (Var 'b') (Var 'd'))
s1 :: Prop
s1 = And (Var 'a') (Var 'd')
s2 :: Prop
s2 = p11
s1s2 :: Prop
```

```
s1s2 = Not (Equiv (s1) (s2))
s3 :: Prop
s3 = And (Var 'b') (Var 'c')
```

1 PropParser

This is my propostional parser using Parsec (parsec2 package)

```
module PropParser where
import Text.ParserCombinators.Parsec   -- :set -ignore-package parsec-3.1.0
import Text.ParserCombinators.Parsec.Expr
import qualified Text.ParserCombinators.Parsec.Token as T
import Text.ParserCombinators.Parsec.Language (haskellDef)
import Text.ParserCombinators.Parsec.Char
import PropChecker
import Char


  -- The Parser
mainParser = do whiteSpace
                e ← expr
                eof
                return e
expr    = buildExpressionParser table term
        <?> "expression"
term0 =    parens expr
        <|> var
        <?> "simple proposition"
term   = do
        t ← term0
        whiteSpace
        return t
table   :: OperatorTable Char () Prop
table   = [[prefix "~" Not]
          ,[binary "&" And AssocLeft, binary "v" Or AssocLeft]
          ,[binary "=>" Imply AssocLeft, binary "<=>" Equiv AssocNone]
          ]
binary name fun assoc = Infix (do {reservedOp name; whiteSpace; return fun}) assoc
prefix  name fun  = Prefix (do {reservedOp name; whiteSpace; return fun})
postfix name fun = Postfix (do {reservedOp name; whiteSpace; return fun})
isVar             :: Char → Bool
isVar c           = isAlpha c ∧ c ≢ 'v'
var               :: Parser Prop
var               = fmap Var $ satisfy isVar
evalProp          :: String → String → Rests → String
evalProp x1 x2 rs
             = case (parse mainParser "" x1) of
                  Left err1 → show err1
                  Right p1 → case (parse mainParser "" x2) of
                                Left err2 → show err2
                                Right p2 → show (propMachine p1 p2 rs)
  -- This will convert the first string to a Prop and allow disagree to work
evalDisagree      :: String → String → Rests → String
```

```
evalDisagree x1 x2 rs = case (parse mainParser "" x1) of
                          Left err1 → show err1
                          Right p1 → case (parse mainParser "" x2) of
                            Left err2 → show err2
                            Right p2 → disagree p1 p2 rs
  -- Restriction parser
getRests             :: [String] → Rests → Either String Rests
getRests [] ps       = Right ps
getRests (r : rs) ps = case (parse mainParser "" r) of
                          Left err → Left "hello"
                          Right p → Right [p]
removeEmpty xs = filter (≢ "") xs
  -- The lexer
lexer      = T.makeTokenParser haskellDef
lexeme     = T.lexeme
parens     = T.parens lexer
natural    = T.natural lexer
reservedOp = T.reservedOp lexer
whiteSpace = T.whiteSpace lexer
```