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

Peter Santos

August 10, 2010

Abstract

This project attempts to address the problem of helping students learn to address problems in natural language using propositional logic. The solution is implemented via an interactive web interface that is developed strictly using the functional programming language Haskell and its relatively new web framework Happstack. The implementation was relatively successful and demonstrates the potential power of Haskell and Happstack in providing educational tools via an interactive web interface.

Introduction

In developing a web interface, Happstack will be used for the web framework, and Blaze-html will be used to generate the html for the web pages. In order to allow for the comparison and evaluation of two logical propositions, three major modules will be needed to make a complete web interface. These three major modules are PropChecker, PropParser, and GetProp, as such the document will be split into three sections describing these modules. PropChecker will take care of the comparing two logical propositions, PropParser will take care of converting a given String from a user to a propositional statement, and GetProp will take care of generating the web interface that will receive propositions from the user and generate custom feedback. The complete web interface will only demonstrate one question that a user can answer at the moment, but it should be noted that generating more questions would simply be a matter of either adding more pages using the exact same layout or adding additional questions with input boxes on the same page. The mean idea is to show that it can be done, and later it can be seen scaling the site with more questions should not be too difficult of a task, once everything is in place.

This first section will describe and explain how the propositional checker works and lay the foundation for how the web interface will generate its results.

The Propositional Checker — PropChecker

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, Imply, etc.

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

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

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

Assoc acts as a 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(t) of variables to bools and find the first bool value(v) 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 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)
```

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 they will be removed 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
```

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, producing a list of 'substitution' tables.

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

The type Rests is meant to represent a list of propositions that another proposition must follow in order to be considered correct.

```
type Rests = [Prop]
```

The function cleanSubst will take a Subst list and only keep the instances where Subst evaluates to true for the given Prop.

```
cleanSubst :: [Subst] → Prop → [Subst]
cleanSubst subs p = filter (flip eval p) subs
```

readySubst will function as cleanSubst, but for multiple restrictions.

```
readySubst              :: [Subst] → Rests → [Subst]
readySubst subs []      = subs
readySubst subs (p : ps) = readySubst (cleanSubst subs p) ps
```

Now that a function that can produce a proper substitution list for a proposition is defined, another function that pairs two substitution lists together needs to be defined. The keepSubst function will take a Subst list and only keep the subsets of the Subst list that is provided.

```
keepSubst :: [Subst] → Set.Set (Char, Bool) → [(Subst, Subst)]
keepSubst subs set1 = [(s, Set.toList set1) | s ← subs, set1 `Set.isSubsetOf` Set.fromList s]
```

finalSubst will create the final substitution list. The first arguments should take a larger 'proper' substitution list and a smaller substitution list. If the substitutions are 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]
```

The function isEquiv works with two propositions that have the same amount of variables.

```
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 first proposition is matched with the first Subst, which always has more variables then the second proposition.

```
isEquiv'              :: Prop → Prop → [(Subst, Subst)] → Bool
isEquiv' p1 p2 subs = and [eval s1 p1 ≡ eval s2 p2 | (s1, s2) ← subs]
```

propMachine is takes two propositions and it's restrictions and determines if they are equivalent. This setup will make it easier to use later on in the web interface.

```
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))
```

The next few functions will deal with producing a helpful custom error message to the user. Arguably, this functions could be put in PropParser instead of here, but since these functions are of a comparing nature, it is felt that its place here would be more appropriate. The first function is needs no explanation.

```
safeHead (h: _) = Just h
safeHead _ =      Nothing
```

fstFalse takes a possible Subst tuple and removes the duplicate entries in order to be used for further processing.

```
fstFalse          :: Maybe (Subst, Subst) → Maybe Subst
fstFalse subPair = case subPair of
                     Just (s1, s2) → Just (rmdups (s1 ++ s2))
                     Nothing → Nothing
```

This function produces the first substitution group in which the two propositions disagree.

```
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
```

Once a single substitution list is generated, disagreeM′ will produce a custom error message describing what substitution instance causes their proposition to be incorrect.

```
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
```

If disagreeM receives Nothing then we can deduce that a given proposition violated the restrictions, since it did not produce a substitution.

```
disagreeM          :: Maybe Subst → String
disagreeM (Just s) = disagreeM′ s
disagreeM Nothing = "your proposition is evaluated, it violates the given " ++
                    "restrictions. "
```

Finally, disagree allows for the generation of a proper error message to be easily implemented in the web interface, using the previously defined functions.

```
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))
```

The next section will describe the parser that is used to convert strings to propositions and evaluate them with given restrictions if they are provided.

The Propositional Parser — PropParser

This propositional parser is built using Parsec, more specifically the parsec2 package, which is meant to be easier to use.

```
module PropParser where
import Text.ParserCombinators.Parsec
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
```

First we define var to correctly generate Prop values.

```
isVar   :: Char → Bool
isVar c = isAlpha c ∧ c ≢ 'v'
var     :: Parser Prop
var     = fmap Var $ satisfy isVar
```

Next, we have the main parser which is made using Expr, a module that is well suited for this purpose, allowing for custom error messages as well as a clean of operators and associativity.

```
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})
```

Next, we have 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
```

This function will be used to in the web interface to collect a user's answer, parse it into a proposition and then compare it to another proposition that will be supplied by the host of the site. Restrictions will need to be supplied in the explicit Prop format.

```
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 function will be used when two propositions are not equivalent. It will generate the correct error message. In this case, the user either violated the restrictions or input a proposition that was not equivalent. It worth noting that improperly formatted input will be caught immediately at the web interface in order to produce a correct error message.

```
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
```

The final section attempts to explain how exactly all the previously defined functions work with the Happstack web framework, as well as the use of the HTML generator Blaze-Html.

The Propositional Checker Website — GetProp

This interactive website uses a modified version of the RqDataUpload.hs source from the Happstack Crash Course at happstack.com. Additional help was received from the people at the happstack IRC channel. In the beginning the important modules to import are of course Happstack.Server and Blaze. Monad is also imported to allow for simplified request matching.

```
 {-# LANGUAGE OverloadedStrings #-}
import Control.Monad          (msum)
import Happstack.Server       (Response, ServerPart,
                               Method (GET, POST), methodM
                               , defaultBodyPolicy, dir, getDataFn
                               , look, lookInput, fileServe, nullDir
                               , notFound
                               , nullConf, ok, simpleHTTP, toResponse
                               , seeOther)
import Text.Blaze             as B
import Text.Blaze.Html4.Strict as B hiding (map)
import Text.Blaze.Html4 ∘ Strict.Attributes as B hiding (dir, title)
import PropChecker
import PropParser as P
```

The advice given is that two pages need to be generated. In Happstack, a ServerPart Response is considered a page. One page will setup the form to collect a proposition from the student and post it, and the next page will parse that information and generate a Response page, which will display whether the proposition is correct or incorrect, and provide feedback if the proposition is malformed or incorrect. To begin, GetProp is executed, the simpleHTTP server takes propcheck, which is a route filter that matches a request to a response.

```
main :: IO ()
main = simpleHTTP nullConf $ propcheck
```

As explained earlier, propcheck is the route filter which matches requests with the correct response. For 'dir "feedback"' has methodM POST attached to it in order match on the specific HTTP request, and if it does match, then produce the page, otherwise a custom error page, errorPage will be generated. This way, anyone trying to make a request on 'feedback' will get the errorPage. This is done with a nested msum (or filter), which allows for a future decision to remove the errorPage or add more possibilities simply by adding or subtracting elements from the nested msum. The next match, dir "static" is used with fileServe to serve up any files that are used, with the last argument stating where to look for the file, in this the current directory. nullDir will check if the path is non-empty, and if it is, the filter will move onto the next item, which is notExist, custom 404 message. This kind of customization can add a sense of user-friendliness.

```
propcheck :: ServerPart Response
propcheck =
  msum [dir "feedback" $ msum [methodM POST ≫ feedback
    , errorPage
    ]
    , dir "static" $ fileServe [] "."
    , nullDir ≫ propForm
    , notExist
    ]
```

This is the form page that will collect the user's input and post it as user_prop if the user clicks on the 'Check this proposition' button. It also acts as the default page. The blaze-html module allows the use of the same tags that would be normally used in an explicit html file.

```
propForm :: ServerPart Response
propForm = ok $ toResponse $
  html $ do
    B.head $ do
      title "Propositional Equivalance Checker"
    B.h1 $ do
      text "Peter's Propositional Equivalence Checker"
    B.body $ do
      p (string $ question1)
      p "Where:"
      ul $ li $ text "a = Room 1 has a lady"
      ul $ li $ text "b = Room 1 has a tiger"
      ul $ li $ text "c = Room 2 has a lady"
      ul $ li $ text "d = Room 2 haa a tiger"
      p "Given:"
      ul $ li $ text "a is not equivalent to b"
      ul $ li $ text "c is not equivalent to d"
    B.div $ do
    B.div $ do
      p (string $ instructions)
      form ! enctype "multipart/form-data" ! B.method "POST" ! action "/feedback" $ do
           input ! type_ "text" ! name "user_prop" ! size "40" ! maxlength "40"
           input ! type_ "submit" ! name "check_prop" ! value "Check this proposition"
           input ! type_ "reset" ! name "clear_prop" ! value "Clear"
instructions :: String
```

```
instructions = "Enter a proposition that describes this situation. "

question1 :: String
question1 = "[...] the king explained to the prisoner that each of the two "
  ++ "rooms contained either a lady or a tiger, but it could be that "
  ++ "there were tigers in both rooms, or ladies in both rooms, or "
  ++ "then again, maybe one room contained a lady and the other room "
  ++ "a tiger."
```

For the feedback page, the getDataFn is used to get the posted value contained in user_prop. The first argument of this function also sets the allow upload file size, but in this case, it doesn't matter what the values are. The function mkBody will post an error message if there is anything wrong with the value that is posted that is not a parsing issue. Once it is determined to be a correct value, in this a case a String, it is sent to the PropParser function evalProp to be evaluated. Based on it's evaluation, an appropriate part of a page will be generated via the isEquivalent function, and put into the body tag of feedback.

```
feedback :: ServerPart Response
feedback =
  do r ← getDataFn (defaultBodyPolicy "/tmp/" 1000 1000 1000) $ look "user_prop"
     ok $ toResponse $
       html $ do
         B.head $ do
           title "Prop Feedback"
         B.h2 $ do
           text "Feedback on given Proposition     "
           img ! src "/static/lambda.gif" ! alt "lambda" ! width "20" ! height "20"
         B.body $ do
           mkBody r
    where
      mkBody (Left errs) =
        do p $ "The following error occurred:"
           mapM_ (p ∘ string) errs
      mkBody (Right theprop) = do
        B.h3 $ do
          text "Analysis"
        isEquivalent (P.evalProp theprop prop1 rests1) theprop
isEquivalent                 :: String → String → Html b
isEquivalent "True" prop  = p (string $ "Your proposition '" ++ prop ++
                                "' correctly describes this situation.")
isEquivalent "False" prop = do
                                p (string $ "Your proposition '" ++ prop ++
                                  "' incorrectly describes this situation.")
                                B.h4 $ text "Here's a tip: "
                                p (string $ "When " ++ evalDisagree prop prop1 rests1 ++
                                  "Carefully look over the " ++
                                  "information that is being given to you and try again.")
isEquivalent error    prop  = do
                                p (string $ ("'" ++ prop ++ "' is not a correctly" ++
                                  " written proposition."))
                                B.h4 $ text "Check "
                                p (string $ betterError error)
betterError     :: String → String
betterError err =           drop 8 (filter (≠ ')') err)
```

The host of the website can enter their proposition and restrictions for each question that is presented.

```
rests1 :: Rests
rests1 = [Not (Equiv (Var 'a') (Var 'c'))
```

```
        , Not (Equiv (Var 'b') (Var 'd'))
        ]
prop1 :: String
prop1 = "(a v b) & (c v d)"
```

These are the custom error pages, one is for when a user tries to go to the feedback page directly, and the other is when they enter a non-existent url. It can clearly be shown that each page must first set its html response code, as can be seen for the first error page where it is just the standard ok and with the second page it is the proper notFound response code.

```
errorPage :: ServerPart Response
errorPage = ok $ toResponse $
  html $ do
    B.head $ do
      title "Error Page"
    B.h1 $ do
      text "Oops!"
    B.body $ do
      p $ "It seems you tried to check your feedback without submitting a proposition."
    B.div $ do
      p $ ""
notExist :: ServerPart Response
notExist = notFound $ toResponse $
  html $ do
    B.head $ do
      title "Error Page"
    B.h1 $ do
      text "Sorry!"
    B.body $ do
      p $ "This page doesn't exist, maybe you can ask for it to be in the next version."
    B.div $ do
      p $ ""
```

Conclusion

This was an enjoyable project, even with all the obstacles encountered, and would have been much smoother with prior knowledge of how web services work, specifically the concepts of server requests and responses as well as the GET and POST concepts. More tutorials, or tutorials of a specific application such as this one would have helped in the learning process, but since this is bleeding edge technology, it is understandable that this wasn't the case. Most of the support in dealing with Happstack came from the happs IRC channel. Which proved to be extremely useful especially during the initial setup of the Happstack framework, as a blur of packages had to be installed due to dependencies and even a reinstallation of Happstack due to faulty previous installations of GHC packages. Another obstacle of working with Happstack was not having a complete solid understanding of Haskell, which made for a more difficult time in deciphering the different types that functions were. Having said that, the code presented does makes sense in this application. Blaze-html also seems very promising, by the looks of it, a person could most likely take any non-complex or possibly complex website and generate it's blaze-html equivalent. The great part about this project was discovering the power of functional programming languages, specifically Haskell. Seeing real applications and how they can be developed on the web and off the web is powerful in demonstrating to students that this language or rather paradigm should not be overlooked when considering application development simply because it is different and not familiar. The support for the Haskell language seems to very lively, especially on the haskell IRC channel. In the future it would be nice to try use Happstack with user accounts and a slightly more complex website. Also, in Haskell functions it would be nice to tie in the application of verification techniques to demonstrate how they would be encountered or used in the real world, even if it is somewhat trivial or unnecessary to do so in this respect to this application. As this project has demonstrated, examples are powerful.