Propositional Logic

Jan van Eijck

Haskell ILLC Project, June 6, 2012

Literate Programming, Again

```
module PropLogic

where

import Data.List
```

A Datatype for Formulas

```
type Name = Int

data Form = Prop Name
          | Neg  Form
          | Cnj [Form]
          | Dsj [Form]
          | Impl Form Form
          | Equiv Form Form
          deriving Eq
```

Displaying Formulas

```
instance Show Form where
  show (Prop x)    = show x
  show (Neg f)     = '-' : show f
  show (Cnj fs)      = "*(" ++ showLst fs ++ ")"
  show (Dsj fs)      = "+(" ++ showLst fs ++ ")"
  show (Impl f1 f2)  = "(" ++ show f1 ++ "==>"
                             ++ show f2 ++ ")"
  show (Equiv f1 f2)  = "(" ++ show f1 ++ "<=>"
                             ++ show f2 ++ ")"

showLst,showRest :: [Form] -> String
showLst [] = ""
showLst (f:fs) = show f ++ showRest fs
showRest [] = ""
showRest (f:fs) = ' ': show f ++ showRest fs
```

Example Formulas

```
p = Prop 1
q = Prop 2
r = Prop 3

form1 = Equiv (Impl p q) (Impl (Neg q) (Neg p))
form2 = Equiv (Impl p q) (Impl (Neg p) (Neg q))
form3 = Impl (Cnj [Impl p q, Impl q r]) (Impl p r)
```

Proposition Letters Occurring in a Formula

```
propNames :: Form -> [Name]
propNames = sort.nub.pnames where
  pnames (Prop name) = [name]
  pnames (Neg f)  = pnames f
  pnames (Cnj fs) = concat (map pnames fs)
  pnames (Dsj fs) = concat (map pnames fs)
  pnames (Impl f1 f2) = concat (map pnames [f1,f2])
  pnames (Equiv f1 f2) =  concat (map pnames [f1,f2])
```

Valuations

```
type Valuation = [(Name,Bool)]

-- all possible valuations for a list of prop letters
genVals :: [Name] -> [Valuation]
genVals [] = [[]]
genVals (name:names) =
  map ((name,True) :) (genVals names)
  ++ map ((name,False):) (genVals names)

-- generate all possible valuations for a formula
allVals :: Form -> [Valuation]
allVals = genVals . propNames
```

Evaluation of Formulas

```
eval :: Valuation -> Form -> Bool
eval [] (Prop c)    =
     error ("no info about " ++ show c)
eval ((i,b):xs) (Prop c)
     | c == i    = b
     | otherwise = eval xs (Prop c)
eval xs (Neg f)  = not (eval xs f)
eval xs (Cnj fs) = all (eval xs) fs
eval xs (Dsj fs) = any (eval xs) fs
eval xs (Impl f1 f2) = not (eval xs f1) || eval xs f2
eval xs (Equiv f1 f2) = eval xs f1 == eval xs f2
```

Satisfiability, Logical Entailment, Equivalence

```
-- f is satisfiable if some valuation makes f true
satisfiable :: Form -> Bool
satisfiable f = any (\ v -> eval v f) (allVals f)

contradiction :: Form -> Bool
contradiction = not . satisfiable

-- logical entailment
entails :: Form -> Form -> Bool
entails f1 f2 = contradiction (Cnj [f1, Neg f2])

-- logical equivalence
equiv :: Form -> Form -> Bool
equiv f1 f2 = entails f1 f2 && entails f2 f1
```

Arrow Free Form

```
-- no precondition: should work for any formula.
arrowfree :: Form -> Form
arrowfree (Prop x) = Prop x
arrowfree (Neg f) = Neg (arrowfree f)
arrowfree (Cnj fs) = Cnj (map arrowfree fs)
arrowfree (Dsj fs) = Dsj (map arrowfree fs)
arrowfree (Impl f1 f2) =
  Dsj [Neg (arrowfree f1), arrowfree f2]
arrowfree (Equiv f1 f2) =
  Dsj [Cnj [f1', f2'], Cnj [Neg f1', Neg f2']]
  where f1' = arrowfree f1
        f2' = arrowfree f2
```