Logic Haskell Exercises

Young W. Lim

2018-09-15 Sat

Outline

1. Based on

2. Logic
   - Using TAMO.hs

Based on

"The Haskell Road to Logic, Maths, and Programming",
K. Doets and J. V. Eijck

Using TAMO.hs

```
module TAMO

where
```

```
:load TAMO
```

Quantifiers as Functions

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :load TAMO
[1 of 1] Compiling TAMO             ( TAMO.hs, interpreted )
Ok, modules loaded: TAMO.
*TAMO> any (>3) [0 ..]
True
*TAMO> any (<3) [0 ..]
True
*TAMO> any (<-1) [0 ..]

<interactive>:5:6:
    parse error on input ‘<-’
    Perhaps this statement should be within a 'do' block?
*TAMO> any (< -1) [0 ..]

^CInterrupted. (takes too long)
```

Predefined Connectives

- Datatype Bool

```
data Bool = False | True
```

- Negation (not)

```
not :: Bool -> Bool
not True = False
not False = True
```

- Conjunction (&&)

```
(&&) :: Bool -> Bool -> Bool
False && x = False
True && x = x
```

- Disjunction (||)

```
(||) :: Bool -> Bool -> Bool
False && x = x
True && x = True
```

Connectives Definition

- Implication (==>)

```
(==>) :: Bool -> Bool -> Bool~
 x ==> y = (not x) || y~
```

- Equivalence (<=>)

```
(<=>) :: Bool -> Bool -> Bool
x <=> y = x == y
```

- Exclusive or (<+>)

```
(<+>) :: Bool -> Bool -> Bool
x <+> y = x /= y
```

Fixity Declaration in Haskell

- `infix` non-associativity
- `infixl` left-associativity
- `infixr`
  right-associativity
- specifies a a precedence
  level from 0 to 9
  - 0 (weakest)
  - 9 (strongest)
  - 10 (normal
    application)

http://zvon.org/other/haskell/Outputsyntax/fixityQdeclaration_reference.html

Fixity Declaration Example

```
main = print (1 +++ 2 *** 3)

infixr  6 +++
infixr  7 ***,///

(+++) :: Int -> Int -> Int
a +++ b = a + 2*b

(***) :: Int -> Int -> Int
a *** b = a - 4*b

(///) :: Int -> Int -> Int
a /// b = 2*a - 3*b


(1 +++ (2 *** 3)) = (1 +++ (2 - 4*3)) = (1 +++ -10) = 1 + 2*(-10) = -19

http://zvon.org/other/haskell/Outputsyntax/fixityQdeclaration_reference.html
```

Fixity of Connectives Definition

- `infix  1 ==>`
- `infix  1 <=>`
- `infixr 2 <+>`

Conncectives Examples

```
*Main> True ==> True
True
*Main> True ==> False
False
*Main> False ==> True
True
*Main> False ==> False
True
----------------------
*Main> (==>) True True
True
*Main> (==>) True False
False
*Main> (==>) False True
True
*Main> (==>) False False
True
```

```
*Main> True <=> True
True
*Main> True <=> False
False
*Main> False <=> True
False
*Main> False <=> False
True
----------------------
*Main> (<=>) True True
True
*Main> (<=>) True False
False
*Main> (<=>) False True
False
*Main> (<=>) False False
True
```

```
*Main> True <+> True
False
*Main> True <+> False
True
*Main> False <+> True
True
*Main> False <+> False
False
------------------------
*Main> (<+>) True True
False
*Main> (<+>) True False
True
*Main> (<+>) False True
True
*Main> (<+>) False False
False
```

Evaluating Conncectives

```
P=True, Q=False

~ P ^ ((P -> Q) <-> ~(Q ^ ~ P))
: T :   T  : F   :  : F : : T
F   :     F      :  :   : F
    :            :  :   F
    :            :  T
    :            F
    F

~ P ^ ((P -> Q) <-> ~(Q ^ ~ P))
F T F   T F  F   F  T F F F T
```

No Multiple Declarations in Haskell

- in ghci, can change the value of a binding

```
P=True
Q=False

P=False
Q=True
```

- in ghc, cannot change the value of a binding

```
Error
Multiple declarations of 'P'
Multiple declarations of 'Q'
```

Evaluating Conncectives in Haskell

```
p = True
q = False

formula1 = (not p) && (p ==> q) <=> not (q && (not p))
                T      T     F            F         T
             F            F                  F
                   F                 T
                               F

formula2 p q = ((not p) && (p ==> q) <=> not (q && (not p)))

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

*Main> formula1                   *Main> :t formula1
False                             formula1 :: Bool
*Main> formula2 True True
False                             Bool type value
*Main> formula2 True False
False                             *Main> :t formula2
*Main> formula2 False True        formula2 :: Bool -> Bool -> Bool
False
*Main> formula2 False False       a function value
True                              taking two Bool type values
                                  returning a Bool type value
```

Validity Check

- True for every possible combination of propositional arguments
- a formula is valid iff it is true under every interpretation
- an argument is valid iff it is impossible for the premises to be true and for the conclusion to be false

List Comprehension

- to enumerate all the possible cases of a truth table
  a list comprehension can be used
- 2 proposition truth table

```
[(p, q)| p <-[True, False], q<-[True, False]]

[(True,True),(True,False),(False,True),(False,False)]
```

- 3 proposition truth table

```
[(p, q, r)| p <-[True, False], q<-[True, False], r<-[True, False]]

[(True,True,True),(True,True,False),(True,False,True),
 (True,False,False),(False,True,True),(False,True,False),
 (False,False,True),(False,False,False)]
```

Boolean Function Types

- Boolean functions with only one variable
  `Bool -> Bool`
- Boolean functions with two variables
  `Bool -> Bool -> Bool`
- Boolean functions with three variables
  `Bool -> Bool -> Bool -> Bool`

Validity Check Functions

- Boolean functions `bf` with only one variable

```
valid1 :: (Bool -> Bool) -> Bool
valid1 bf =  (bf True) && (bf False)
```

- Boolean functions `bf` with two variables

```
valid2 :: (Bool -> Bool -> Bool)  -> Bool
valid2 bf =   (bf True  True)
           && (bf True  False)
           && (bf False True)
           && (bf False False)
```

Validity Check Functions using list comprehensions

- Boolean functions bf with two variables

```
valid2 :: (Bool -> Bool -> Bool)  -> Bool
valid2 bf = and [ bf p q  | p <- [True,False],
                            q <- [True,False]]
```

- Boolean functions bf with three variables

```
valid3 :: (Bool -> Bool -> Bool -> Bool) -> Bool
valid3 bf = and [ bf p q r | p <- [True,False],
                             q <- [True,False],
                             r <- [True,False]]
```

- Boolean functions bf with only one variable

```
valid4 :: (Bool -> Bool -> Bool -> Bool -> Bool) -> Bool
valid4 bf = and [ bf p q r s | p <- [True,False],
                               q <- [True,False],
                               r <- [True,False],
                               s <- [True,False]]
```

Validity Function Examples

- a valid function with only one variable
  $P \vee \neg P$: tautology

```
excluded_middle :: Bool -> Bool
excluded_middle p = p || not p
```

- a valid function with two variables
  $P \Rightarrow (Q \Rightarrow P)$: tautology
  $(P \Rightarrow Q) \Rightarrow P$: not valid $(P \Rightarrow Q) \supset P$

```
form1 p q = p ==> (q ==> p)
form2 p q = (p ==> q) ==> p
```

```
----------------------------------------------------------
*Main> valid1 excluded_middle
True
*Main> valid2 form1
True
*Main> valid2 form2
False
```

Logical Equivalence Check Function Types

- check logical equivalence for propositonal functions with 1 parameter

```
logEquiv1 ::  (Bool -> Bool) ->  -- bf1 function type
              (Bool -> Bool) ->  -- bf2 function type
               Bool
```

- check logical equivalence for propositonal functions with 2 parameters

```
logEquiv2 :: (Bool -> Bool -> Bool) ->  -- bf1 function type
             (Bool -> Bool -> Bool) ->  -- bf2 function type
              Bool
```

- check logical equivalence for propositonal functions with 2 parameters

```
logEquiv3 :: (Bool -> Bool -> Bool -> Bool) -> -- bf1 function type
             (Bool -> Bool -> Bool -> Bool) -> -- bf2 function type
              Bool
```

Logical Equivalence Check Function Definitions

- check logical equivalence for propositonal functions with 1 parameter

```
logEquiv1 bf1 bf2 =  (bf1 True  <=> bf2 True) &&
                     (bf1 False <=> bf2 False)
```

- check logical equivalence for propositonal functions with 2 parameters

```
logEquiv2 bf1 bf2  = and [(bf1 p q) <=> (bf2 p q) | p <- [True,False],
                                                    q <- [True,False]]
```

- check logical equivalence for propositonal functions with 3 parameters

```
logEquiv3 bf1 bf2 = and [(bf1 p q r) <=> (bf2 p q r) | p <- [True,False],
                                                       q <- [True,False],
                                                       r <- [True,False]]
```

Logical Equivalence Examples

- $q \oplus q = F$
- $p \oplus F = p$
- $(p \oplus q) \oplus q = p \oplus (q \oplus q) = p \oplus F = p$

```
formula3 p q = p

formula4 p q = (p <+> q) <+> q

formula5 p q = p <=> ((p <+> q) <+> q)

-------------------------------------------------------
*Main> logEquiv2 formula3 formula4  -- logical equivalent
True
*Main> valid2 formula5  -- tautology
True
```

Instances of a class

- the Monad class and its instance
- Monad : an abstract data type
- Monad m : a paramerized data type
- Maybe Int : a concrete type
- methods *declared* in the class definition (return and >>=)
- *implemented* (*defined*) in the instance definition (return and >>=)

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

instance Monad Maybe where
   return   x       =  Just x
   Nothing  >>= _   =  Nothing
   (Just x) >>= f   =  f x
```

Instances of the class TF (1)

- instances of the TF class must implement
  - validity checking method
  - logical equivalence checking method
- instance TF Bool
  - Bool in instance TF Bool matches
  - p in class TF p

```
class TF p where
  valid :: p -> Bool
  lequiv :: p -> p -> Bool

instance TF Bool
 where
  valid  = id
  lequiv f g = f == g
```

Instances of the class TF (2)

- `instance TF p => TF (Bool -> p)`
  - `p` matches `p` in `class TF p`
  - `TF p` is an instance of `class TF p`
- `TF (Bool -> p)`
  - `(Bool -> p)` matches `p` in `class TF p`
  - `TF (Bool -> p)` is an instance of `class TF p`

```
instance TF p => TF (Bool -> p)
 where
  valid f     = valid (f True) && valid (f False)
  lequiv f g = (f True)  `lequiv` (g True) &&
               (f False) `lequiv` (g False)
```

Constraint Kind

- constraints appear in types to the left of the => arrow
- have a very restricted syntax. They can only be:
  - Class constraints, e.g. Show a
  - Implicit parameter constraints, e.g. ?x::Int (with the -XImplicitParams flag)
  - Equality constraints, e.g. a ~ Int (with the -XTypeFamilies or -XGADTs flag)

1. law of double negation

- $P \equiv \neg\neg P$

```
test1  = lequiv id (\ p -> not (not p))
```

2. laws of idempotence

- $P \wedge P \equiv P$
- $P \vee P \equiv P$

```
test2a = lequiv id (\ p -> p && p)
test2b = lequiv id (\ p -> p || p)
```

3. laws of implication

- $(P \Rightarrow Q) \equiv \neg P \vee Q$
- $\neg(P \Rightarrow Q) \equiv P \wedge \neg Q$

```
test3a = lequiv (\ p q -> p ==> q) (\ p q -> not p || q)
test3b = lequiv (\ p q -> not (p ==> q)) (\ p q -> p && not q)
```

4. laws of contraposition

- $(\neg P \Rightarrow \neg Q) \equiv Q \Rightarrow P$
- $(P \Rightarrow \neg Q) \equiv Q \Rightarrow \neg P$
- $(\neg P \Rightarrow Q) \equiv \neg Q \Rightarrow P$

```
test4a = lequiv (\ p q -> not p ==> not q) (\ p q -> q ==> p)
test4b = lequiv (\ p q -> p ==> not q) (\ p q -> q ==> not p)
test4c = lequiv (\ p q -> not p ==> q) (\ p q -> not q ==> p)
```

5. laws of biconditions

- $(P \Leftrightarrow Q) \equiv ((P \Rightarrow Q) \wedge (Q \Rightarrow P))$
- $(P \Leftrightarrow Q) \equiv ((\neg P \vee Q) \wedge (\neg QP))$

```
test5a = lequiv (\ p q -> p <=> q)
                (\ p q -> (p ==> q) && (q ==> p))
test5b = lequiv (\ p q -> p <=> q)
                (\ p q -> (p && q) || (not p && not q))
```

6. laws of commutativity

- $P \wedge Q \equiv Q \wedge P$
- $P \vee Q \equiv Q \vee P$

```
test6a = lequiv (\ p q -> p && q) (\ p q -> q && p)
test6b = lequiv (\ p q -> p || q) (\ p q -> q || p)
```

7. De Morgan's laws

- $\neg(P \wedge Q) \equiv \neg P \neg Q$
- $\neg(P \vee Q) \equiv \neg P \neg Q$

```
test7a = lequiv (\ p q -> not (p && q))
                (\ p q -> not p || not q)
test7b = lequiv (\ p q -> not (p || q))
                (\ p q -> not p && not q)
```

8. laws of asscociativity

- $P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$
- $P \vee (Q \vee R) \equiv (P \vee Q) \vee R$

```
test8a = lequiv (\ p q r -> p && (q && r))
                (\ p q r -> (p && q) && r)
test8b = lequiv (\ p q r -> p || (q || r))
                (\ p q r -> (p || q) || r)
```

9. law of distribution

- $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
- $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$

```
test9a = lequiv (\ p q r -> p && (q || r))
                (\ p q r -> (p && q) || (p && r))
test9b = lequiv (\ p q r ->  p || (q && r))
                (\ p q r -> (p || q) && (p || r))
```