Functional Programming for Logicians - Lecture 2

Type Classes, QuickCheck, Modal Logic

Malvin Gattinger

5 June 2018

```haskell
module L2 where

import E1S
import Data.List
import Test.QuickCheck
```

Overview

- Recap: What we did yesterday
- `isValid` for Propositional Logic
- Polymorphism
- Type Classes
- QuickCheck
- Example: Modal Logic
- `hlint` and `ghc -Wall`

What we did yesterday

- Functions: `f . g` and $
- Lists: `map`, comprehension, ++, `!!`
- Recursion
- Lambdas: `(\x -> x + x)`
- Guards, Pattern Matching
- Propositional Logic in Haskell
- Exercises:
  - list functions
  - prime numbers
  - propositional Logic

Recap: Propositional Logic

```
data Form = P Integer | Neg Form | Conj Form Form
  deriving (Eq,Ord,Show)

type Assignment = [Integer]

satisfies :: Assignment -> Form -> Bool
satisfies v (P k)      = k `elem` v
satisfies v (Neg f)    = not (satisfies v f)
satisfies v (Conj f g) = satisfies v f && satisfies v g

varsIn :: Form -> [Integer]
varsIn (P k)      = [k]
varsIn (Neg f)    = varsIn f
varsIn (Conj f g) = nub (varsIn f ++ varsIn g)
```

```haskell
allAssignmentsFor :: [Integer] -> [Assignment]
allAssignmentsFor []     = [ [] ]
allAssignmentsFor (p:ps) =
  [ p:rest | rest <- allAssignmentsFor ps ]
  ++ allAssignmentsFor ps

isValid :: Form -> Bool
isValid f =
  and [ v `satisfies` f  | v <- allAssignmentsFor (varsIn f) ]
```

Examples:

```
λ> isValid $ P 1
False
λ> isValid $ Neg (Conj (P 1) (Neg (P 1)))
True
```

type, data, newtype

- `type` is for abbreviations:

```
type Person = (String,Integer)
```

- `data` is for new stuff:

```
data Form = P Int | Neg Form | Conj Form Form
```

- `newtype` is for single-case new stuff that actually abbreviates:

```
newtype Name = Name String
```

curry and uncurry

```
λ> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
λ> :t uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
λ> uncurry (+) (7,5)
12
```

⇒ Board exercise: Define `curry` and `uncurry`!

Polymorphism

A fancy name for something you already know: Functions can be defined for any type, using type variables like `a` and `b` here:

```
λ> :t fst
fst :: (a, b) -> a
λ> :t map
map :: (a -> b) -> [a] -> [b]
```

Polymorphism

A fancy name for something you already know: Functions can be defined for any type, using type variables like `a` and `b` here:

```
λ> :t fst
fst :: (a, b) -> a
λ> :t map
map :: (a -> b) -> [a] -> [b]
```

Note that partial application of `map` already determines the type:

```
λ> :t map (++ " omg!")
map (++ " omg!") :: [String] -> [String]
```

Whenever you write `map` it is fixed at compile-time what `a` is!

Type classes

Some functions are polymorphic, but not totally. For example, we can only `lookup` something if we know how to check for equality:

```
λ> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Type classes

Some functions are polymorphic, but not totally. For example, we can only `lookup` something if we know how to check for equality:

```
λ> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

`Eq` is a *type class* defined like this:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Type classes

Some functions are polymorphic, but not totally. For example, we can only `lookup` something if we know how to check for equality:

```
λ> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

`Eq` is a *type class* defined like this:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Suppose we have:

```
data Animal = Cat | Horse | Bird
```

Then `Cat == Horse` is not defined until we make a new *instance* of `Eq` to teach Haskell when two animals can are equal ...

When are Animals equal?

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Animal where
  (==) Cat   Cat   = True
  (==) Horse Horse = True
  (==) Bird  Bird  = True
  (==) _     _     = False
```

The Ord class

```
λ> :i Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  {-# MINIMAL compare | (<=) #-}
```

```
instance Ord Animal where
  (<=) _     Horse = True
  (<=) Cat   Cat   = True
  (<=) Bird  _     = True
  (<=) _     _     = False
```

Note that it is our job to make <= reflexive and transitive!

The Show class

```haskell
class Show a where
  show :: a -> String

instance Show Animal where
  show Cat    = "Cat"
  show Horse = "Horse"
  show Bird  = "Bird"
```

The Show class

```haskell
class Show a where
  show :: a -> String

instance Show Animal where
  show Cat    = "Cat"
  show Horse  = "Horse"
  show Bird   = "Bird"
```

Convention: `show x` should return valid Haskell code.

It is not meant for pretty printing!

```haskell
prettyPrint :: Animal -> String

prettyPrint Cat    = "🐦"

prettyPrint Horse  = "🐈"

prettyPrint Bird   = "🐎"
```

Deriving Instances

All of this was a bit tedious and trivial, so let GHC do it:

```
data Animal = Cat | Horse | Bird  deriving (Eq,Ord,Show)
```

A non-trivial instance example: sets

```
λ> [1,1,3] == [1,3,3]
False
λ> [6,1] == [1,6]
False
```

```haskell
newtype Set a = Set [a]
```

A non-trivial instance example: sets

```
λ> [1,1,3] == [1,3,3]
False
λ> [6,1] == [1,6]
False
```

```haskell
newtype Set a = Set [a]

instance (Ord a) => Eq (Set a) where
  (==) (Set xs) (Set ys) = sort (nub xs) == sort (nub ys)


instance (Ord a, Show a) => Show (Set a) where
  show (Set xs) = "Set " ++ show (sort (nub xs))
```

```
λ> Set [1,1,3] == Set [1,3,3]
True
λ> Set [1,1,3]
Set [1,3]
λ> Set [6,1] == Set [1,6]
True
```

Type Class overview

- `Eq` — stuff where == works
- `Show` — stuff that can be `shown`
- `Ord` — stuff `that can be compared and sorted`

Kinds

Expressions like `Int`, `Maybe`, `Show` do not have a type, but a *kind*:

```
λ> :k Int
Int :: *
λ> :k Maybe
Maybe :: * -> *
λ> :k Show
Show :: * -> Constraint
λ> :k Set
Set :: * -> *
λ> :k Either
Either :: * -> * -> *
```

Think of kinds as "meta-types": The kind of something tells you whether something is a type or what it does to types.

A Maybe Either Example

```
lookupTwo :: Eq a => a -> [(a,b)]
                      -> [(a,c)]
                      -> Maybe (Either b c)
lookupTwo x one two =
  case (lookup x one, lookup x two) of
    (Just y , _     ) -> Just (Left y)
    (Nothing, Just z) -> Just (Right z)
    _                 -> Nothing
```

QuickCheck

Properties

Let `a` be some type.

Then `a -> Bool` is the type of `a` properties.

Properties can be used for *testing*.

Example: Quicksort

Quicksort is a very efficient sorting algorithm.

Here is an implementation in Haskell:

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort [ a | a <- xs, a <= x ]
                ++ [x]
                ++ quicksort [ a | a <- xs, a > x ]
```

The `quicksort` function should turn *any* finite list of items into an *ordered* list of items.

(This is not the true real Quicksort(TM) . . .)

We can check if a list is ordered like this:

```
isOrdered :: Ord a => [a] -> Bool
isOrdered []      = True
isOrdered (x:xs) = all (>= x) xs && isOrdered xs
```

The `QuickCheck` library allows us to do the following:

```
quickCheck (\xs -> isOrdered (quicksort xs::[Int]))
```

To see what it does, use `verboseCheck` instead.

Here's another property we want:

```
sameLength :: [Int] -> [Int] -> Bool
sameLength xs ys = length xs == length ys
```

```
quickCheck (\xs -> sameLength xs (quicksort xs))
```

Does it hold?

Here's another property we want:

```haskell
sameLength :: [Int] -> [Int] -> Bool
sameLength xs ys = length xs == length ys
```

```haskell
quickCheck (\xs -> sameLength xs (quicksort xs))
```

Does it hold?

→ See also:

- Juan Pedro Villa: A QuickCheck Tutorial: Generators
- Hackage documentation: Test.QuickCheck

`QuickChecking` our Porpositional Logic

Can we do this?

```
quickCheck (\f -> isValid f == isValid (Neg (Neg f)))
```

Hint: Not yet. 😉

Teaching QuickCheck some Logic

```haskell
myAtoms :: [Integer]
myAtoms = [1..5]

instance Arbitrary Form where
  arbitrary = sized randomForm where
    randomForm :: Int -> Gen Form
    randomForm 0 = P <$> elements myAtoms
    randomForm n = oneof
      [ P <$> elements myAtoms
      , Neg <$> randomForm (n `div` 2)
      , Conj <$> randomForm (n `div` 2)
             <*> randomForm (n `div` 2) ]
```

Now we can do:

```haskell
verboseCheck (\f -> isValid f == isValid (Neg (Neg f)))
```

Which other properties do we expect to hold?

QuickCheck as a Research Tool

0. Have a conjecture about X.
1. Implement X in Haskell.
2. Implement an `Arbitrary` instance for X.
3. Formulate conjecture as a property.
4. `quickCheck`

QuickCheck as a Research Tool

0. Have a conjecture about X.
1. Implement X in Haskell.
2. Implement an `Arbitrary` instance for X.
3. Formulate conjecture as a property.
4. `quickCheck`

?. Profit

Modal Logic

Kripke models and modal formulas

```
type Proposition = Int

type World = Integer
type Universe = [World]
type Valuation = World -> [Proposition]
type Relation = [(World,World)]
data KripkeModel = KrM Universe Valuation Relation

data ModForm = Prp Proposition
             | Not ModForm
             | Con ModForm ModForm
             | Box ModForm
```

Modal Logic Semantics

```
makesTrue :: (KripkeModel,World) -> ModForm -> Bool
makesTrue (KrM _ v _, w) (Prp k)   = k `elem` v w
makesTrue (m,w)          (Not f)   = not (makesTrue (m,w) f)
makesTrue (m,w)          (Con f g) =
  makesTrue (m,w) f && makesTrue (m,w) g
makesTrue (KrM u v r, w) (Box f)   =
  all (\w' -> makesTrue (KrM u v r,w') f) ws where
    ws = [ y | y <- u, (w,y) `elem` r ]
```

(Side remark: If you are also annoyed that we have to repeat the definitions for propositional logic here, check out "final tagless" interpreters, see http://okmij.org/ftp/tagless-final/)

Modal Logic: Example

```haskell
myModel :: KripkeModel
myModel = KrM [0,1,2] myVal myRel where
  myVal 0 = [1,2]
  myVal 1 = [1]
  myVal 2 = [1,3]
  myVal _ = undefined
  myRel = [(0,0),(0,1),(0,2)]
```

```
λ> (myModel,0) `makesTrue` Box (Prp 1)
True
λ> (myModel,0) `makesTrue` Box (Prp 2)
False
```

🥗 🍝

See you again at *13:00* in F3.20.