Functional Programming for Logicians - Lecture 2

Type Classes, QuickCheck, Modal Logic

Malvin Gattinger

5 June 2018
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: \( f \circ g \) and \( \$ \)
- Lists: map, comprehension, ++, !!
- Recursion
- Lambdas: \((\lambda x \to x + x)\)
- Guards, Pattern Matching
- Propositional Logic in Haskell
- Exercises:
  - list functions
  - prime numbers
  - propositional Logic
Recap: Propositional Logic

\[
data \text{ Form } = \text{P Integer } | \text{ Neg Form } | \text{ Conj Form Form}
\]

\[
deriving (\text{Eq,Ord,Show})
\]

\[
type \text{ Assignment } = [\text{Integer}]
\]

\[
satisfies :: \text{Assignment} \to \text{Form} \to \text{Bool}
\]

\[
satisfies v (\text{P k}) = k `\text{elem}` v
\]

\[
satisfies v (\text{Neg f}) = \text{not} (\text{satisfies v f})
\]

\[
satisfies v (\text{Conj f g}) = \text{satisfies v f} \&\& \text{satisfies v g}
\]

\[
\text{varsIn :: Form } \to [\text{Integer}]
\]

\[
\text{varsIn (P k)} = [k]
\]

\[
\text{varsIn (Neg f)} = \text{varsIn f}
\]

\[
\text{varsIn (Conj f g)} = \text{nub (varsIn f ++ varsIn g)}
\]
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:

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

- **data** is for new stuff:

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

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

```haskell
newtype Name = Name String
```
curry and uncurry

\[\lambda \text{> } :t \text{ curry}\]
\[\text{curry} \::\: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c\]
\[\lambda \text{> } :t \text{ uncurry}\]
\[\text{uncurry} \::\: (a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c\]
\[\lambda \text{> uncurry } (+) \text{ (7,5)}\]
\[12\]

⇒ 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:

\[
\text{\textbackslash{}> \\texttt{:t fst}}
\]
\[
\text{fst :: (a, b) \rightarrow a}
\]

\[
\text{\textbackslash{}> \\texttt{:t map}}
\]
\[
\text{map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]}
\]
A fancy name for something you already know: Functions can be defined for any type, using type variables like `a` and `b` here:

\[
\lambda> :t \text{fst} \\
\text{fst} :: (a, b) \to a \\
\lambda> :t \text{map} \\
\text{map} :: (a \to b) \to [a] \to [b]
\]

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

\[
\lambda> :t \text{map (}++ \ " \text{omg!}\text{)}\) \\
\text{map (}++ \ " \text{omg!}\text{)} :: [\text{String}] \to [\text{String}]
\]

Whenever you write `map` it is fixed at compile-time what `a` is!
Some functions are polymorphic, but not totally. For example, we can only lookup something if we know how to check for equality:

\[
\lambda > :t \text{lookup} \\
\text{lookup} :: \text{Eq } a \Rightarrow a \to [(a, b)] \to \text{Maybe } b
\]
Type classes

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

\[
\lambda > \text{t \ lookup}
\]

\[
\text{lookup} :: \text{Eq } a \Rightarrow a \rightarrow [(a, b)] \rightarrow \text{Maybe } b
\]

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:

lambda: `:t lookup`

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

Eq is a type class defined like this:

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

Suppose we have:

```haskell
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

\$\lambda>: i \text{ Ord}\$

class Eq a => Ord a where

\[\text{compare} :: a \to a \to \text{Ordering}\]

\[\text{(<)} :: a \to a \to \text{Bool}\]

\[\text{(<=)} :: a \to a \to \text{Bool}\]

\[\text{(>) :: a \to a \to \text{Bool}}\]

\[\text{(>=) :: a \to a \to \text{Bool}}\]

\[\text{max :: a \to a \to a}\]

\[\text{min :: a \to a \to a}\]

\{-\# \text{MINIMAL compare | (<=) \#-}\}

\text{instance Ord Animal where}

\[\text{(<=)} _ \_ \text{Horse} = \text{True}\]

\[\text{(<=)} \text{Cat} \text{Cat} = \text{True}\]

\[\text{(<=)} \text{Bird} _ _ = \text{True}\]

\[\text{(<=)} _ _ = \text{False}\]

Note that it is our job to make \(<=\) reflexive and transitive!
The Show class

class Show a where
    show :: a -> String

instance Show Animal where
    show Cat    = "Cat"
    show Horse  = "Horse"
    show Bird   = "Bird"
The Show class

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!

prettyPrint :: Animal -> String

prettyPrint Cat  = " 🐱"
prettyPrint Horse = " 🐴"
prettyPrint Bird  = " 🦅"
All of this was a bit tedious and trivial, so let GHC do it:

```haskell
data Animal = Cat | Horse | Bird deriving (Eq, Ord, Show)
```
A non-trivial instance example: sets

\lambda> [1,1,3] == [1,3,3]
False
\lambda> [6,1] == [1,6]
False

newtype Set a = Set [a]
A non-trivial instance example: sets

\lambda> [1,1,3] == [1,3,3]
False
\lambda> [6,1] == [1,6]
False

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))

\lambda> Set [1,1,3] == Set [1,3,3]
True
\lambda> Set [1,1,3]
Set [1,3]
\lambda> 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:

\[
\begin{align*}
\lambda & : k \text{ Int} \\
\text{Int} & :: * \\
\lambda & : k \text{ Maybe} \\
\text{Maybe} & :: * \to * \\
\lambda & : k \text{ Show} \\
\text{Show} & :: * \to \text{Constraint} \\
\lambda & : k \text{ Set} \\
\text{Set} & :: * \to * \\
\lambda & : k \text{ Either} \\
\text{Either} & :: * \to * \to *
\end{align*}
\]

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

```haskell
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
Let a be some type.

Then a \( \rightarrow \) Bool is the type of a properties.

Properties can be used for testing.
**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:

\[
\text{isOrdered} :: \text{Ord}\ a \Rightarrow [a] \to \text{Bool}
\]

\[
\text{isOrdered} \ [\] = \text{True}
\]

\[
\text{isOrdered} \ (x:xs) = \text{all} \ (\geq x) \ xs \land \text{isOrdered} \ xs
\]

The QuickCheck library allows us to do the following:

\[
\text{quickCheck} \ (\lambda xs \rightarrow \text{isOrdered} \ (\text{quicksort} \ xs::[\text{Int}])))
\]

To see what it does, use \text{verboseCheck} instead.
Here’s another property we want:

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

Does it hold?

→ See also:

- Juan Pedro Villa: A QuickCheck Tutorial: Generators
- Hackage documentation: Test.QuickCheck
QuickChecking our Propositional Logic

Can we do this?

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

Hint: Not yet. 😊
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:

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
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

\[
\text{myModel} :: \text{KripkeModel}
\]
\[
\text{myModel} = \text{KrM} [0,1,2] \text{ myVal myRel where}
\]
\[
\text{myVal} 0 = [1,2]
\]
\[
\text{myVal} 1 = [1]
\]
\[
\text{myVal} 2 = [1,3]
\]
\[
\text{myVal} _ = \text{undefined}
\]
\[
\text{myRel} = [(0,0),(0,1),(0,2)]
\]

\[
\lambda> (\text{myModel},0) `\text{makesTrue}` \text{ Box (Prp 1)}
\]
True

\[
\lambda> (\text{myModel},0) `\text{makesTrue}` \text{ Box (Prp 2)}
\]
False
See you again at 13:00 in F3.20.