Informatics 1
Functional Programming Lectures 13 and 14
Monday 10 and Tuesday 11 November 2014

Type Classes

Don Sannella
University of Edinburgh
Mock exam

Slots and rooms have now been assigned
Mon 17–Fri 21 November
Check the course web page for your assignment

You may only come to the assigned slot and room
The 2014 Informatics 1 Competition

- First prize: A bottle of champagne or equivalent in book tokens. And glory!
- Previous year entries are online:
  www.inf.ed.ac.uk/teaching/courses/inf1/fp/#competition
- Number of prizes depend on number/quality of entries.
- Sponsored by Galois (galois.com)
- Send code and image(s), list everyone who contributed.
- E-mail your entry to Karoliina Lehtinen
  <M.K.Lehtinen@sms.ed.ac.uk>
- **Deadline: midnight, Wednesday 19 November 2014**
Part I

Type classes
Element

\[\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}\]

-- comprehension
\[
\text{elem } x \text{ ys} \quad = \quad \text{or } [\ x == y \mid y \leftarrow \text{ys} ]
\]

-- recursion
\[
\text{elem } x \ [\] \quad = \quad \text{False}
\]
\[
\text{elem } x \ (y:ys) \quad = \quad x == y \ || \ \text{elem } x \ ys
\]

-- higher-order
\[
\text{elem } x \ \text{ys} \quad = \quad \text{foldr } (||) \ \text{False} \ (\text{map } (x ==) \ \text{ys})
\]
Using element

*Main> elem 1 [2,3,4]
False

*Main> elem 'o' "word"
True

*Main> elem (1,'o') [(0,'w'),(1,'o'),(2,'r'),(3,'d')]
True

*Main> elem "word" ["list","of","word"]
True

*Main> elem (\x -> x) [(\x -> -x), (\x -> -(-x))]
No instance for (Eq (a -> a)) arising from a use of 'elem'
Possible fix: add an instance declaration for (Eq (a -> a))
Equality type class

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

instance Eq Int where
  (==) = eqInt

instance Eq Char where
  x == y = ord x == ord y

instance (Eq a, Eq b) => Eq (a,b) where
  (u,v) == (x,y) = (u == x) && (v == y)

instance Eq a => Eq [a] where
  [] == [] = True
  [] == y:ys = False
  x:xs == [] = False
  x:xs == y:ys = (x == y) && (xs == ys)
Element, translation

\[
\text{data} \ \text{EqDict} \ a \ = \ \text{EqD} \ (a \rightarrow a \rightarrow \text{Bool})
\]

\[
eq \ :: \ \text{EqDict} \ a \rightarrow a \rightarrow a \rightarrow \text{Bool}
eq (\text{EqDict} \ f) \ = \ f
\]

\[
elem \ :: \ \text{EqD} \ a \rightarrow a \rightarrow [a] \rightarrow \text{Bool}
\]

\[
\text{-- comprehension}
\]
\[
elem \ d \ x \ ys \ = \ \text{or} \ [ \ eq \ d \ x \ y \mid y <- ys ]
\]

\[
\text{-- recursion}
\]
\[
elem \ d \ x \ [] \ = \ \text{False}
elem \ d \ x \ (y:ys) \ = \ eq \ d \ x \ y \ || \ elem \ x \ ys
\]

\[
\text{-- higher-order}
\]
\[
elem \ d \ x \ ys \ = \ \text{foldr} \ (||) \ \text{False} \ (\text{map} \ (eq \ d \ x) \ ys)
\]
Type classes, translation

\texttt{dInt} \quad :: \quad \text{EqDict} \quad \text{Int}
\texttt{dInt} \quad = \quad \text{EqD} \quad \text{eqInt}

\texttt{dChar} \quad :: \quad \text{EqDict} \quad \text{Char}
\texttt{dChar} \quad = \quad \text{EqD} \quad f
where
\texttt{f} \quad \texttt{x} \quad \texttt{y} \quad = \quad \texttt{eq} \quad \texttt{dInt} \quad \texttt{(ord x)} \quad \texttt{(ord y)}

\texttt{dPair} \quad :: \quad (\text{EqDict} \quad \texttt{a}, \text{EqDict} \quad \texttt{b}) \quad \rightarrow \quad \text{EqDict} \quad (\texttt{a},\texttt{b})
\texttt{dPair} \quad (\texttt{da},\texttt{db}) \quad = \quad \text{EqD} \quad f
where
\texttt{f} \quad (\texttt{u},\texttt{v}) \quad (\texttt{x},\texttt{y}) \quad = \quad \texttt{eq} \quad \texttt{da} \quad \texttt{u} \quad \texttt{x} \quad \&\& \quad \texttt{eq} \quad \texttt{db} \quad \texttt{v} \quad \texttt{y}

\texttt{dList} \quad :: \quad \text{EqDict} \quad \texttt{a} \quad \rightarrow \quad \text{EqDict} \quad [\texttt{a}]
\texttt{dList} \quad \texttt{d} \quad = \quad \text{EqD} \quad f
where
\texttt{f} \quad [\quad] \quad [\quad] \quad = \quad \texttt{True}
\texttt{f} \quad [\quad] \quad (\texttt{y}:\texttt{ys}) \quad = \quad \texttt{False}
\texttt{f} \quad (\texttt{x}:\texttt{xs}) \quad [\quad] \quad = \quad \texttt{False}
\texttt{f} \quad (\texttt{x}:\texttt{xs}) \quad (\texttt{y}:\texttt{ys}) \quad = \quad \texttt{eq} \quad \texttt{d} \quad \texttt{x} \quad \texttt{y} \quad \&\& \quad \texttt{eq} \quad (\texttt{dList} \quad \texttt{d}) \quad \texttt{xs} \quad \texttt{ys}
Using element, translation

```haskell
*Main> elem dInt 1 [2,3,4]
False

*Main> elem dChar 'o' "word"
True

*Main> elem (dPair dInt dChar) (1,'o') [(0,'w'),(1,'o')]
True

*Main> elem (dList dChar) "word" ["list","of","word"]
True
```

Haskell uses types to write code for you!
Part II

Eq, Ord, Show
Eq, Ord, Show

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

  -- minimum definition: (==)
  x /= y = not (x == y)

class (Eq a) => Ord a where
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool

  -- minimum definition: (<=)
  x < y = x <= y && x /= y
  x > y = y < x
  x >= y = y <= x

class Show a where
  show :: a -> String
Part III

Booleans, Tuples, Lists
Instances for booleans

```haskell
instance Eq Bool where
    False == False = True
    False == True  = False
    True  == False = False
    True  == True  = True

instance Ord Bool where
    False <= False  = True
    False <= True  = True
    True  <= False = False
    True  <= True  = True

instance Show Bool where
    show False    = "False"
    show True     = "True"
```
Instances for pairs

**instance** (Eq a, Eq b) => Eq (a,b) **where**
(x,y) == (x’,y’) = x == x’ && y == y’

**instance** (Ord a, Ord b) => Ord (a,b) **where**
(x,y) <= (x’,y’) = x < x’ || (x == x’ && y <= y’)

**instance** (Show a, Show b) => Show (a,b) **where**
show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"
Instances for lists

instance  Eq  a  =>  Eq  [a]  where
          []    ==    []    =    True
          []    ==    y:ys  =    False
          x:xs  ==    []    =    False
          x:xs  ==    y:ys  =    x == y && xs == ys

instance  Ord  a  =>  Ord  [a]  where
          []    <=    ys    =    True
          x:xs  <=    []    =    False
          x:xs  <=    y:ys  =    x < y || (x == y && xs <= ys)

instance  Show  a  =>  Show  [a]  where
    show    []       =    "[]"
    show    (x:xs)  =    "[" ++ showSep x xs ++ "]"
          where
        showSep x    []      =    show x
        showSep x    (y:ys)  =    show x ++ "," ++ showSep y ys
Deriving clauses

```
data Bool = False | True
    deriving (Eq, Ord, Show)

data Pair a b = MkPair a b
    deriving (Eq, Ord, Show)

data List a = Nil | Cons a (List a)
    deriving (Eq, Ord, Show)
```

Haskell uses types to write code for you!
Part IV

Sets, revisited
Sets, revisited

\[
\text{instance } \text{Ord } a \Rightarrow \text{Eq } (\text{Set } a) \text{ where } \\
\quad s == t = s \text{ `equal` } t
\]

Note that this differs from the derived instance!
Part V

Numbers
Numerical classes

```haskell
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  fromInteger :: Integer -> a
  -- minimum definition: (+), (-), (*), fromInteger
  negate x = fromInteger 0 - x

class (Num a) => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  -- minimum definition: (/), fromRational
  recip x = 1/x

class (Num a, Ord a) => Real a where
  toRational :: a -> Rational

class (Real a, Enum a) => Integral a where
  div, mod :: a -> a -> a
  toInteger :: a -> Integer
```
A built-in numerical type

```
instance Num Float where
  (+)       = builtInAddFloat
  (-)       = builtInSubtractFloat
  (*)       = builtInMultiplyFloat
  negate    = builtInNegateFloat
  fromInteger = builtInFromIntegerFloat

instance Fractional Float where
  (/)       = builtInDivideFloat
  fromRational = builtInFromRationalFloat
```
module Natural (Nat) where

import Test.QuickCheck

data Nat = MkNat Integer

invariant :: Nat -> Bool
invariant (MkNat x) = x >= 0

instance Eq Nat where
  MkNat x == MkNat y = x == y

instance Ord Nat where
  MkNat x <= MkNat y = x <= y

instance Show Nat where
  show (MkNat x) = show x
instance Num Nat

where

MkNat x + MkNat y = MkNat (x + y)

MkNat x - MkNat y
  | x >= y = MkNat (x - y)
  | otherwise = error (show (x-y) ++ " is negative")

MkNat x * MkNat y = MkNat (x * y)

fromInteger x
  | x >= 0 = MkNat x
  | otherwise = error (show x ++ " is negative")

negate = undefined
Natural.hs (3)

\[
\text{prop \_ plus :: Integer \rightarrow Integer \rightarrow Property}
\]
\[
\text{prop \_ plus \ m \ n =}
\]
\[
(m \geq 0) \land (n \geq 0) \implies (m+n \geq 0)
\]

\[
\text{prop \_ times :: Integer \rightarrow Integer \rightarrow Property}
\]
\[
\text{prop \_ times \ m \ n =}
\]
\[
(m \geq 0) \land (n \geq 0) \implies (m\times n \geq 0)
\]

\[
\text{prop \_ minus :: Integer \rightarrow Integer \rightarrow Property}
\]
\[
\text{prop \_ minus \ m \ n =}
\]
\[
(m \geq 0) \land (n \geq 0) \land (m \geq n) \implies (m-n \geq 0)
\]
module NaturalTest where

import Natural

m, n :: Nat
m = fromInteger 2
n = fromInteger 3
Test run

ghci NaturalTest
Ok, modules loaded: NaturalTest, Natural.
*NaturalTest> m
2
*NaturalTest> n
3
*NaturalTest> m+n
5
*NaturalTest> n-m
1
*NaturalTest> m-n
*** Exception: -1 is negative
*NaturalTest> m*n
6
*NaturalTest> fromInteger (-5) :: Nat
*** Exception: -5 is negative
*NaturalTest> MkNat (-5)
Not in scope: data constructor ‘MkNat’
Hiding—the secret of abstraction

module Natural (Nat) where ...

> ghci NaturalTest
*NaturalTest> let m = fromInteger 2
*NaturalTest> let s = fromInteger (-5)
*** Exception: -5 is negative
*NaturalTest> let s = MkNat (-5)
Not in scope: data constructor ‘MkNat’

VS.

module NaturalUnabs (Nat (MkNat)) where ...

> ghci NaturalUnabs
*NaturalUnabs> let p = MkNat (-5) -- breaks invariant
*NaturalUnabs> invariant p
False
Part VI

Seasons
Seasons

```haskell
data Season = Winter | Spring | Summer | Fall

next :: Season -> Season
next Winter = Spring
next Spring = Summer
next Summer = Fall
next Fall = Winter

warm :: Season -> Bool
warm Winter = False
warm Spring = True
warm Summer = True
warm Fall = True
```
Eq, Ord

```haskell
instance Eq Seasons where
  Winter == Winter   = True
  Spring == Spring   = True
  Summer == Summer   = True
  Fall  == Fall      = True
  _     == _         = False

instance Ord Seasons where
  Spring <= Winter   = False
  Summer <= Winter   = False
  Summer <= Spring   = False
  Fall  <= Winter    = False
  Fall  <= Spring    = False
  Fall  <= Summer    = False
  _     <= _         = True

instance Show Seasons where
  show Winter   = "Winter"
  show Spring   = "Spring"
  show Summer   = "Summer"
  show Fall     = "Fall"
```
Class Enum

class Enum a where
  toEnum :: Int -> a
  fromEnum :: a -> Int
  succ, pred :: a -> a
  enumFrom :: a -> [a] -- [x..]
  enumFromTo :: a -> a -> [a] -- [x..y]
  enumFromThen :: a -> a -> [a] -- [x,y..]
  enumFromThenTo :: a -> a -> a -> [a] -- [x,y..z]

  -- minimum definition: toEnum, fromEnum
  succ x = toEnum (fromEnum x + 1)
pred x = toEnum (fromEnum x - 1)
enumFrom x = map toEnum [fromEnum x ..]
enumFromTo x y = map toEnum [fromEnum x .. fromEnum y]
enumFromThen x y = map toEnum [fromEnum x, fromEnum y ..]
enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
Syntactic sugar

-- [x..]    =  enumFrom x
-- [x..y]    =  enumFromTo x y
-- [x,y..]   =  enumFromThen x y
-- [x,y..z]  =  enumFromThenTo x y z
Enumerating Int

```haskell
instance Enum Int where
  toEnum x = x
  fromEnum x = x
  succ x = x+1
  pred x = x-1
  enumFrom x = iterate (+1) x
  enumFromTo x y = takeWhile (<= y) (iterate (+1) x)
  enumFromThen x y = iterate (+y-x) x
  enumFromThenTo x y z = takeWhile (<= z) (iterate (+y-x) x)

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs) | p x = x : takeWhile p xs
  | otherwise = []
```
Enumerating Seasons

```haskell
instance Enum Seasons where

    fromEnum Winter    = 0
    fromEnum Spring    = 1
    fromEnum Summer    = 2
    fromEnum Fall      = 3

    toEnum 0           = Winter
    toEnum 1           = Spring
    toEnum 2           = Summer
    toEnum 3           = Fall
```
Deriving Seasons

```
data Season = Winter | Spring | Summer | Fall
  deriving (Eq, Ord, Show, Enum)
```

Haskell uses types to write code for you!
Seasons, revisited

next :: Season -> Season
next x = toEnum ((fromEnum x + 1) `mod` 4)

warm :: Season -> Bool
warm x = x `elem` [Spring .. Fall]

-- [Spring .. Fall] = [Spring, Summer, Fall]
Part VII

Shape
Shape

type  Radius  =  Float

type  Width   =  Float

type  Height  =  Float

data  Shape   =  Circle  Radius
 |  Rect  Width  Height

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect w h) = w * h
Eq, Ord, Show

instance Eq Shape where
  Circle r == Circle r' = r == r'
  Rect w h == Rect w' h' = w == w' && h == h'
  _ == _ = False

instance Ord Shape where
  Circle r <= Circle r' = r < r'
  Circle r <= Rect w' h' = True
  Rect w h <= Rect w' h' = w < w' || (w == w' && h <= h')
  _ <= _ = False

instance Show Shape where
  show (Circle r) = "Circle " ++ showN r
  show (Radius w h) = "Radius " ++ showN w ++ " " ++ showN h

showN :: (Num a) => a -> String
showN x | x >= 0 = show x
         | otherwise = "(" ++ show x ++ ")"
Deriving Shapes

```haskell
data Shape   =  Circle Radius
               |  Rect Width Height
deriving    (Eq, Ord, Show)
```

Haskell uses types to write code for you!
Part VIII

Expressions
Expression Trees

```haskell
data Exp  =  Lit Int
            | Exp :+: Exp
            | Exp :*: Exp

eval :: Exp -> Int
eval (Lit n)  =  n
eval (e :+: f)  =  eval e + eval f
eval (e :*: f)  =  eval e * eval f
```

```haskell
*Main> eval (Lit 2 :+: (Lit 3 :*: Lit 3))
11
*Main> eval ((Lit 2 :+: Lit 3) :*: Lit 3)
15
```
Eq, Ord, Show

instance  Eq Exp  where
  Lit n  ==  Lit n'  =  n == n'
  e :+: f  ==  e' :+: f'  =  e == e' && f == f'
  e :*: f  ==  e' :*: f'  =  e == e' && f == f'
  _  ==  _  =  False

instance  Ord Exp  where
  Lit n  <=  Lit n'  =  n < n'
  Lit n  <=  e' :+: f'  =  True
  Lit n  <=  e' :*: f'  =  True
  e :+: f  <=  e' :+: f'  =  e < e' || (e == e' && f <= f')
  e :+: f  <=  e' :*: f'  =  True
  e :*: f  <=  e' :*: f'  =  e < e' || (e == e' && f <= f')
  _  <=  _  =  False

instance  Show Exp  where
  show (Lit n)  =  "Lit " ++ showN n n
  show (e :+: f)  =  "(" ++ show e ++ " :+: " ++ show f ++ ")"
  show (e :*: f)  =  "(" ++ show e ++ " :* : " ++ show f ++ ")"
Deriving Expressions

```haskell
data Exp = Lit Int |
        Exp :+: Exp |
        Exp :+: Exp |
    deriving (Eq, Ord, Show)
```

Haskell uses types to write code for you!