

Informatics 1

Functional Programming Lectures 15 and 16

Monday 14 and Tuesday 15 November 2011

# Type Classes

Philip Wadler

University of Edinburgh

# Mock exam

Sign up for a slot

Any afternoon, Monday 21–Friday 25 November

# Extra tutorial

4–5pm Wednesday 16 November 2011

Appleton Tower, room 4.12

Do a previous class or final exam and bring your answer

# Revision session

Tomorrow, Tuesday 15 November 2011

What would you like me to revise?

Say in class, or after, or send e-mail

# The 2011 Informatics 1 Competition

- Prize: A bottle of champagne or equivalent in book tokens. And glory!
- Previous year entries are online:

<http://www.inf.ed.ac.uk/teaching/courses/inf1/fp/#competition>

- In 2009 we had the following prizes:
  - 1 × first prize: £20
  - 2 × second prize: £10
  - 4 × third prize: £5
- Sponsored by Galois (galois.com)
- List everyone who worked on the entry.
- E-mail your entry to to Phil Scott <phil.scott@ed.ac.uk>
- *Deadline: 4pm Wednesday 23 November 2011*

Part I

Type classes

# Element

```
elem :: Eq a => a -> [a] -> Bool

-- comprehension
elem x ys      =  or [ x == y | y <- ys ]

-- recursion
elem x []      =  False
elem x (y:ys) =  x == y || elem x ys

-- higher-order
elem x ys      =  foldr (||) False (map (x ==) 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))
```

# 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

```
data EqDict a      = EqD (a -> a -> Bool)
```

```
eq                :: EqDict a -> a -> a -> Bool  
eq (EqDict f)    = f
```

```
elem :: EqD a -> a -> [a] -> Bool
```

```
-- comprehension
```

```
elem d x ys      = or [ eq d x y | y <- ys ]
```

```
-- recursion
```

```
elem d x []      = False
```

```
elem d x (y:ys) = eq d x y || elem x ys
```

```
-- higher-order
```

```
elem d x ys      = foldr (||) False (map (eq d x) ys)
```

# Type classes, translation

```
dInt      :: EqDict Int
dInt      = EqD eqInt
```

```
dChar     :: EqDict Char
dChar     = EqD f
  where
    f x y  = eq dInt (ord x) (ord y)
```

```
dPair     :: (EqDict a, EqDict b) -> EqDict (a,b)
dPair (da,db) = EqD f
  where
    f (u,v) (x,y) = eq da u x && eq db v y
```

```
dList     :: EqDict a -> EqDict [a]
dList d   = EqD f
  where
    f [] []      = True
    f [] (y:ys)  = False
    f (x:xs) []  = False
    f (x:xs) (y:ys) = eq d x y && eq (dList d) xs ys
```

## Using element, translation

```
*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

```
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

```
instance Ord a => Eq (Set a) where  
  s == t    = s `equal` t
```

Note that this differs from the derived instance!

Part V

Numbers

# Numerical classes

```
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

# Natural.hs (1)

```
module Natural(Nat) where  
import Test.QuickCheck  
  
newtype 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
```

## Natural.hs (2)

```
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 (fromInteger x)
```

```
    | otherwise       = error (show x ++ " is negative")
```

```
negate                = undefined
```

## Natural.hs (3)

```
prop_plus :: Integer -> Integer -> Property
prop_plus m n =
  (m >= 0) && (n >= 0) ==> (m+n >= 0)
```

```
prop_times :: Integer -> Integer -> Property
prop_plus m n =
  (m >= 0) && (n >= 0) ==> (m*n >= 0)
```

```
prop_minus :: Integer -> Integer -> Property
prop_minus m n =
  (m >= 0) && (n >= 0) && (m >= n) ==> (m-n >= 0)
```

# NaturalTest.hs

```
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

```
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

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

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

# Show

```
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

```
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

```
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

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

Haskell uses types to write code for you!

## Part VIII

# Expressions

# Expression Trees

```
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
```

```
*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
show (e :+: f)    = "(" ++ show e ++ ":+:" ++ show f ++ ")"
show (e :* f)     = "(" ++ show e ++ ":*:" ++ show f ++ ")"
```

# Deriving Expressions

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

Haskell uses types to write code for you!

Part IX

Micro-Haskell