Informatics 1

Functional Programming Lecture 15 and 16

Monday 17 and Tuesday 18 November 2008

Type Classes

Philip Wadler

University of Edinburgh

The 2008 Informatics 1 Competition

- Prize: A bottle of champagne or book token equivalent
- Sponsored by Galois (galois.com)
- Deadline: 4pm Friday 28 November 2008
- You may find some inspiration here:

  www.contextfreeart.org

  (Thanks to Aleksandar Krastev for the suggestion.)

Required reading

Haskell: The Craft of Functional Programming, Second Edition,
Simon Thompson, Addison-Wesley, 1999.

Thompson, Chapters 1–3 (pp. 1–52): by Mon 29 Sep 2008.
Thompson, Chapters 4–5 (pp. 53–95): by Mon 6 Oct 2008.
Thompson, Chapters 6–7 (pp. 96–134): by Mon 13 Oct 2008.
Thompson, Chapters 8–9 (pp. 135–166): by Mon 20 Oct 2008.
Thompson, Chapters 10–11 (pp. 167–209): by Mon 3 Nov 2008.
Thompson, Chapters 12–14 (pp. 210–279): by Mon 10 Nov 2008.
Thompson, Chapters 15–17 (pp. 280–382): by Mon 17 Nov 2008.
Thompson, Chapters 18–20 (pp. 338–441): by Mon 24 Nov 2008.

Thompson and other books available in ITO.

Part I

Type classes and translation

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
  (==)  =  eqChar

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) x y  =  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 eqChar

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

Equality over functions

Equality over functions

```
class Small a where
  forAll :: (a -> Bool) -> Bool

instance Small Char where
  forAll p  =  and [ p x | x <- ['\000'..'\255'] ]

instance Small Bool where
  forAll p  =  and [ p x | x <- [False, True] ]

instance (Small a, Small b) => Small (a,b) where
  forAll p  =  forAll (\x -> (forAll (\y -> p (x,y))))

instance (Small a, Eq b) => Eq (a -> b) where
  f == g  =  forAll (\x -> f x == g x)

*Main elem (\x -> x) [(\x -> not x), (\x -> not (not x))]
True
```

Part III

Boolean

Boolean

```
data Bool = False | True

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

(&&) :: Bool -> Bool -> Bool
False && q   =  False
True  && q   =  q

(||) :: Bool -> Bool -> Bool
False || q   =  q
True  || q   =  True
```

Type classes

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

  x /= y  =  not (x == y)

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

  x <= y  =  x < y || x == y
  x > y   =  y < x
  x >= y  =  y <= x

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

Boolean with deriving clause

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

Derived instances

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

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

instance Show Bool where
  show False    =  "False"
  show True     =  "True"
```

Part IV

Tuples and lists

Tuples

```
data (a,b) = (a,b) deriving (Eq,Ord,Show)

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 ++ ")"
```

Lists

```
data [a] = [] | a:[a] deriving (Eq,Ord,Show)

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
  []     <  []     =  False
  []     <  y: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
```

Part V

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
```

Seasons with deriving clause

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

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

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

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

The Enum class

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

  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

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)
```

Derived instance

```
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
```

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 VI

Shape

Shape

```
type  Radius  =  Float
type  Width   =  Float
type  Height  =  Float

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

area :: Shape -> Float
area (Circle r)  =  pi * r^2
area (Rect w h)  =  w * h
```

Derived instances

```
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 ++ ")"
```

Part VII

Expressions

Expression Trees

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

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
```

Derived instances

```
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')

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

Part VIII

Numbers

Numerical classes

```
class (Eq a, Show a) => Num a where
  (+),(-),(*)    ::  a -> a -> a
  negate         ::  a -> a
  fromInteger    ::  Integer -> a

class (Num a) => Fractional a where
  (/)            ::  a -> a -> a
  recip          ::  a -> a
  fromRational   ::  Rational -> a

  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 class

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

class Fractional Float where
  (/)            =  builtInDivideFloat
  fromRational   =  builtInFromRationalFloat
```

Points

```
data  Point  =  Pnt Float Float

scalar :: Float -> Point
scalar x  =  Pnt x x

instance Num Point where
  Pnt x y + Pnt x' y'  =  Pnt (x+x') (y+y')
  Pnt x y - Pnt x' y'  =  Pnt (x-x') (y-y')
  Pnt x y * Pnt x' y'  =  Pnt (x*x') (y*y')
  negate (Pnt x y)     =  Pnt (-x) (-y)
  fromInteger z        =  scalar (fromInteger z)

class Fractional Point where
  Pnt x y / Pnt x' y'  =  Pnt (x/x') (y/y')
  fromRational z       =  scalar (fromRational z)
```

Points

```
instance Eq Point where
  Pnt x y == Pnt x′ y′  =  x == x′ && y == y′

instance Ord Point where
  Pnt x y <  Pnt x′ y′  =  x < x′ && y < y′

glb,lub :: Point -> Point -> Point
Pnt x y `glb` Pnt x′ y′  =  Pnt (x `min` x′) (y `min` y′)
Pnt x y `lub` Pnt x′ y′  =  Pnt (x `max` x′) (y `max` y′)
```

Part IX

Speeding up append

How long does it take to append?

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys  =  []
(x:xs) ++ ys  =  x:(xs ++ ys)

   "abcd" ++ "ef"
=  'a':("bcd" ++ "ef")
=  'a':('b':("cd" ++ "ef"))
=  'a':('b':('c':("d" ++ "ef")))
=  'a':('b':('c':('d':("" ++ "ef"))))
=  'a':('b':('c':('d':"ef")))
=  "abcdef"
```

How long does it take to append?

Associate to the right

```
"a"++("b"++("c"++("d"++("e"++[]))))   =   "abcde"
```

$1+1+1+1+1 = 5$ steps

Appending n strings of length one: $1+\cdots+1 = n$ steps

Associate to the left

```
(((([]++"a")++"b")++"c")++"d")++"e")   =   "abcde"
```

$0+1+2+3+4 = 10$ steps

Appending n strings of length one: $0+1+\cdots+(n-1) = (n-1)\times n/2$ steps

A tricky way to improve the speed

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x  =  f (g x)

type ShowS = String -> String

showString :: String -> ShowS
showString s t  =  s ++ t

   (showString x . showString y) []
=  showString x (showString y [])
=  x ++ (y ++ [])
```

A tricky way to improve the speed

```
    (showString "a" .
     (showString "b" .
      (showString "c" .
       (showString "d" .
        (showString "e"))))) []
=   (((((showString "a") .
          showString "b") .
           showString "c") .
            showString "d") .
             showString "e") []
=   (showString "a"
       (showString "b"
          (showString "c"
             (showString "d"
                (showString "e" [])))))
=   "abcde"
```

Takes n steps regardless!

The show class

```
class Show a where
  showsPrec :: Int -> a -> ShowS
  show      :: a -> String

  showsPrec d x s   =   show x ++ s
  show x            =   showsPrec 0 x ""

shows        :: (Show a) => a -> ShowS
shows x s   =   show x ++ s
```

Part X

Precedence

Operator Precedence

```
infixl 9  !!
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `div`, `mod`, `rem`, `quot`
infixl 6  +, -
infixr 5  :, ++
infix  4  ==, /=, <, <=, >, >=, `elem`, `notElem`
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 0  $, $!, `seq`
```

Expressions Trees, with precedence

```
infixl 6 :+:
infixl 7 :*:

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

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
```

Expression Trees, derived type

```
instance Show Exp where
  showsPrec d (Lit n) s
    =  showParen (d > 10)
         (showString "Lit " .
          showsPrec 11 n)
  showsPrec d (e :+: f) s
    =  showParen (d > 6)
         (showsPrec 7 e .
          showString " :+: " .
          showsPrec 7 f)
  showsPrec d (e :*: f) s | d <= 7
    =  showParen (d > 7)
         (showsPrec 8 e .
          showString " :*: " .
          showPrec 8 f)
```