Informatics 1

Functional Programming Lecture 11 and 12

Monday 2 and Tuesday 3 November 2009

Algebraic Data Types

Philip Wadler

University of Edinburgh

Reminders

- Tutorial groups for students requiring extra help
  Contact Tamise Totterdell at ITO to join
- Study group for students desiring more challenging work
- Use the newsgroups!
- Staff-Student Liaison Meeting

Part I

Review

Differences, four ways

Comprehension

```
differences x ys  =  [ x-y | y <- ys, x>y ]
```

Higher-order functions, nested scope

```
differences x ys  =  map sub (filter gtr ys)
  where
  sub y  =  x-y
  gtr y  =  x>y
```

Higher-order functions, lambda expressions

```
differences x ys  =  map (\y -> x-y) (filter (\y -> x>y) ys)
```

Higher-order functions, sections

```
differences x ys  =  map (x-) (filter (x>) ys)
```

List comprehension with two qualifiers

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

List comprehension with two qualifiers—binding

```
f n = [ (i,j) | i <- [1..n], j <- [i..n] ]
```

```
*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Binding occurrence

Bound occurrence

Scope of binding

List comprehension with two qualifiers—binding

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Binding occurrence

Bound occurrence

Scope of binding

List comprehension with two qualifiers—binding

```
f n  =  [ (i,j) | i <- [1..n], j <- [i..n] ]

*Main> f 3
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Binding occurrence

Bound occurrence

Scope of binding

Defining list comprehensions

$$
\begin{array}{lcl}
[\, e \mid x \leftarrow [l_1, \ldots, l_n],\, q \,] & = & [\, e_x^{l_1} \mid q_x^{l_1} \,] \mathbin{++} \cdots \mathbin{++} [\, e_x^{l_n} \mid q_x^{l_n} \,] \\
[\, e \mid \texttt{True},\, q \,] & = & [\, e \mid q \,] \\
[\, e \mid \texttt{False},\, q \,] & = & [\,] \\
[\, e \mid \bullet \,] & = & [\, e \,]
\end{array}
$$

Evaluating a list comprehension

```
    [ (i,j) | i <- [1..3], j <- [i..3] ]
=
    [ (1,j) | j <- [1..3] ] ++
    [ (2,j) | j <- [2..3] ] ++
    [ (3,j) | j <- [3..3] ]
=
    [(1,1)|•] ++ [(1,2)|•] ++ [(1,3)|•] ++
                 [(2,2)|•] ++ [(2,3)|•] ++
                              [(3,3)|•]
=
    [(1,1)] ++ [(1,2)] ++ [(1,3)] ++
               [(2,2)] ++ [(2,3)] ++
                          [(3,3)]
=
    [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Another example

```
    [ (i,j) | i <- [1..3], j <- [1..3], i <= j ]
=
    [ (1,j) | j <- [1..3], 1 <= j ] ++
    [ (2,j) | j <- [1..3], 2 <= j ] ++
    [ (3,j) | j <- [1..3], 3 <= j ]
=
    [(1,1)|1<=1] ++ [(1,2)|1<=2] ++ [(1,3)|1<=3] ++
    [(2,1)|2<=1] ++ [(2,2)|2<=2] ++ [(2,3)|2<=3] ++
    [(3,1)|3<=1] ++ [(3,2)|3<=2] ++ [(3,3)|3<=3]
=
    [(1,1)|•] ++ [(1,2)|•] ++ [(1,3)|•] ++
    []            ++ [(2,2)|•] ++ [(2,3)|•] ++
    []            ++ []           ++ [(3,3)|•]
=
    [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Translating list comprehensions

$$
\begin{array}{lcl}
[\, e \mid x \leftarrow l,\, q \,] & = & \texttt{concat}\,(\texttt{map}\,(\lambda x.\,[\, e \mid q \,])\,l) \\
[\, e \mid b,\, q \,] & = & \texttt{if}\; b\; \texttt{then}\; [\, e \mid q \,]\; \texttt{else}\; [\,] \\
[\, e \mid \bullet \,] & = & [\, e \,]
\end{array}
$$

Part II

Algebraic types

Everything is an algebraic type

```
data Bool = False | True
data Season = Winter | Spring | Summer | Fall
data Shape = Circle Float | Rectangle Float Float
data Exp = Lit Int | Add Exp Exp | Mul Exp Exp
data List a = Nil | Cons a (List a)
data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
data Maybe a = Nothing | Just a
data Pair a b = Pair a b
data Sum a b = Left a | Right b
data Nat = Zero | Succ Nat
```

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
```

Boolean — eq and show

```
eqBool :: Bool -> Bool -> Bool
eqBool False False  =  True
eqBool False True   =  False
eqBool True  False  =  False
eqBool True  True   =  True

showBool :: Bool -> String
showBool False  =  "False"
showBool True   =  "True"
```

Part IV

Seasons

Seasons

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

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

Seasons—eq and show

```
eqSeason :: Season -> Season -> Bool
eqSeason Winter Winter  =  True
eqSeason Spring Spring  =  True
eqSeason Summer Summer  =  True
eqSeason Fall   Fall    =  True
eqSeason x      y       =  False

showSeason :: Season -> String
showSeason Winter  =  "Winter"
showSeason Spring  =  "Spring"
showSeason Summer  =  "Summer"
showSeason Fall    =  "Fall"
```

Seasons and integers

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

toInt :: Season -> Int
toInt Winter  =  0
toInt Spring  =  1
toInt Summer  =  2
toInt Fall    =  3

fromInt :: Int -> Season
fromInt 0  =  Winter
fromInt 1  =  Spring
fromInt 2  =  Summer
fromInt 3  =  Fall

next :: Season -> Season
next x  =  fromInt ((toInt x + 1) `mod` 4)

eqSeason :: Season -> Season -> Bool
eqSeason x y  =  (toInt x == toInt y)
```

Part V

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
```

Shape—eq and show

```
eqShape :: Shape -> Shape -> Bool
eqShape (Circle r) (Circle r')    =  (r == r')
eqShape (Rect w h) (Rect w' h')   =  (w == w') && (h == h')
eqShape x          y              =  False

showShape :: Shape -> String
showShape (Circle r)  =  "Circle " ++ showF r
showShape (Rect w h)  =  "Rect " ++ showF w ++ " " ++ showF h

showF :: Float -> String
showF x | x >= 0      =  show x
        | otherwise   =  "(" ++ show x ++ ")"
```

Part VI

Lists

Lists

With names

```
data  List a  =  Nil
              |  Cons a (List a)

append :: List a -> List a -> List a
append Nil ys           =  ys
append (Cons x xs) ys   =  Cons x (append xs ys)
```

With built-in notation

```
data  [a]  =  []
           |  a : [a]

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

Part VII

Expression Trees

Expression Trees

```
data Exp  =  Lit Int
          |  Add Exp Exp
          |  Mul Exp Exp

evalExp :: Exp -> Int
evalExp (Lit n)    =  n
evalExp (Add e f)  =  evalExp e + evalExp f
evalExp (Mul e f)  =  evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)    =  show n
showExp (Add e f)  =  par (showExp e ++ "+" ++ showExp f)
showExp (Mul e f)  =  par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s  =  "(" ++ s ++ ")"
```

Expression Trees

```
e0, e1 :: Exp
e0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
e1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)

*Main> showExp e0
"(2+(3*3))"

*Main> evalExp e0
11

*Main> showExp e1
"((2+3)*3)"

*Main> evalExp e1
15
```

Expression Trees, Infix

```
data  Exp  =  Lit Int
           |  Exp `Add` Exp
           |  Exp `Mul` Exp

evalExp :: Exp -> Int
evalExp (Lit n)    =  n
evalExp (e `Add` f)  =  evalExp e + evalExp f
evalExp (e `Mul` f)  =  evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)       =  show n
showExp (e `Add` f)  =  par (showExp e ++ "+" ++ showExp f)
showExp (e `Mul` f)  =  par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s  =  "(" ++ s ++ ")"
```

Expression Trees, Infix

```
e0, e1 :: Exp
e0 = Lit 2 `Add` (Lit 3 `Mul` Lit 3)
e1 = (Lit 2 `Add` Lit 3) `Mul` Lit 3

*Main> showExp e0
"(2+(3*3))"

*Main> evalExp e0
11

*Main> showExp e1
"((2+3)*3)"

*Main> evalExp e1
15
```

Expression Trees, Symbols

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

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

showExp :: Exp -> String
showExp (Lit n)     =  show n
showExp (e :+: f)   =  par (showExp e ++ "+" ++ showExp f)
showExp (e :*: f)   =  par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s  =  "(" ++ s ++ ")"
```

Expression Trees, Symbols

```
e0, e1 :: Exp
e0 = Lit 2 :+: (Lit 3 :*: Lit 3)
e1 = (Lit 2 :+: Lit 3) :*: Lit 3

*Main> showExp e0
"(2+(3*3))"

*Main> evalExp e0
11

*Main> showExp e1
"((2+3)*3)"

*Main> evalExp e1
15
```

Part VIII

Propositions

Propositions

```
type Name = String
data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :|: Prop
          | Prop :&: Prop
          deriving (Eq, Ord)

type Names = [Name]
type Env = [(Name,Bool)]
```

Showing a proposition

```
showProp :: Prop -> String
showProp (Var x)    =  x
showProp (F)        =  "F"
showProp (T)        =  "T"
showProp (Not p)    =  par ("~" ++ showProp p)
showProp (p :|: q) =  par (showProp p ++ "|" ++ showProp q)
showProp (p :&: q) =  par (showProp p ++ "&" ++ showProp q)

par :: String -> String
par s  =  "(" ++ s ++ ")"
```

Names in a proposition

```
names :: Prop -> Names
names (Var x)     =  [x]
names (F)         =  []
names (T)         =  []
names (Not p)     =  names p
names (p :|: q)   =  nub (names p ++ names q)
names (p :&: q)   =  nub (names p ++ names q)
```

Evaluating a proposition in an environment

```
eval :: Env -> Prop -> Bool
eval e (Var x)          =  lookUp e x
eval e (F)              =  False
eval e (T)              =  True
eval e (Not p)          =  not (eval e p)
eval e (p :|: q)        =  eval e p || eval e q
eval e (p :&: q)        =  eval e p && eval e q

lookUp :: Eq a => [(a,b)] -> a -> b
lookUp xys x  =  the [ y | (x',y) <- xys, x == x' ]
  where
  the [x]  =  x
```

Propositions

```
p0 :: Prop
p0 =  (Var "a" :&: Var "b") :|:
      (Not (Var "a") :&: Not (Var "b"))

e0 :: Env
e0 =  [("a",False), ("b",False)]

*Main> show p0
(a & b) | (~a & ~b)

*Main> names p0
["a","b"]

*Main> eval e0 p0
True

*Main> lookUp e0 "a"
False
```

All possible environments

```
envs :: Names -> [Env]
envs []       =  [[]]
envs (x:xs)   =  [ (x,False):e | e <- envs xs ] ++
                 [ (x,True ):e | e <- envs xs ]
```

Alternative

```
envs :: Names -> [Env]
envs []       =  [[]]
envs (x:xs)   =  [ (x,b):e | b <- bs, e <- envs xs ]
  where
  bs = [False,True]
```

All possible environments

```
   envs []
=  [[]]

   envs ["b"]
=  [("b",False):[]] ++ [("b",True ):[]]
=  [[("b",False)],
    [("b",True )]]

   envs ["a","b"]
=  [("a",False):e | e <- envs ["b"] ] ++
   [("a",True ):e | e <- envs ["b"] ]
=  [("a",False):[("b",False)],("a",False):[("b",True )]] ++
   [("a",True ):[("b",False)],("a",True ):[("b",True )]]
=  [[("a",False),("b",False)],
    [("a",False),("b",True )],
    [("a",True ),("b",False)],
    [("a",True ),("b",True )]]
```

Satisfiable

```
satisfiable :: Prop -> Bool
satisfiable p  =  or [ eval e p | e <- envs (names p) ]
```

Propositions

```
p0 :: Prop
p0 =  (Var "a" :&: Var "b") :|:
      (Not (Var "a") :&: Not (Var "b"))

*Main> envs (names p0)
[[("a",False),("b",False)],
 [("a",False),("b",True)],
 [("a",True ),("b",False)],
 [("a",True ),("b",True )]]

*Main> [ eval e p0 | e <- envs (names p0) ]
[True,
 False,
 False,
 True]

*Main> satisfiable p0
True
```

Aside: all subslists of a list

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

Aside: all subslists of a list

```
    subs []
=   [[]]

    subs ["b"]
=   subs [] ++ ["b":ys | ys <- subs []]
=   [[]] ++ ["b":[]]
=   [[], ["b"]]

    subs ["a","b"]
=   subs ["b"] ++ ["a":ys | ys <- subs ["b"]]
=   [[], ["b"]] ++ ["a":[], "a":["b"]]
=   [[], ["b"], ["a"], ["a","b"]]
```