Informatics 1

Functional Programming Lecture 9 and 10

Monday 28 and Tuesday 29 October 2013

Algebraic Data Types

Don Sannella

University of Edinburgh

Part I

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 List a = Nil | Cons a (List a)
data Nat = Zero | Succ Nat
data Exp = Lit Int | Add Exp Exp | Mul Exp Exp
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
```

Part II

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 III

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 IV

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 ++ ")"
```

Shape—tests and selectors

```
isCircle :: Shape -> Bool
isCircle (Circle r)   =  True
isCircle (Rect w h)   =  False

isRect :: Shape -> Bool
isRect (Circle r)   =  False
isRect (Rect w h)   =  True

radius :: Shape -> Float
radius (Circle r)   =  r

width :: Shape -> Float
width (Rect w h)   =  w

height :: Shape -> Float
height (Rect w h)   =  h
```

Shape—pattern matching

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

area :: Shape -> Float
area s  =
  if isCircle s then
     let
        r = radius s
     in
        pi * r^2
  else if isRect s then
     let
        w = width s
        h = height s
     in
        w * h
  else error "impossible"
```

Part V

Lists

Lists

With declarations

```
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

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

Part VI

Natural numbers

Naturals

With names

```
data  Nat   =   Zero
            |   Succ Nat

power :: Float -> Nat -> Float
power x Zero        =   1.0
power x (Succ n)    =   x * power x n
```

With built-in notation

```
(^^) :: Float -> Int -> Float
x ^^ 0   =   1.0
x ^^ n   =   x * (x ^^ (n-1))
```

Naturals

With declarations

```
add :: Nat -> Nat -> Nat
add m Zero       =  m
add m (Succ n)   =  Succ (add m n)

mul :: Nat -> Nat -> Nat
mul m Zero       =  Zero
mul m (Succ n)   =  add (mul m n) m
```

With built-in notation

```
(+) :: Int -> Int -> Int
m + 0  =  m
m + n  =  (m + (n-1)) + 1

(*) :: Int -> Int -> Int
m * 0  =  0
m * n  =  (m * (n-1)) + m
```

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" :&: Not (Var "a"))

e0 :: Env
e0 =  [("a",True)]

*Main> showProp p0
(a&(~a))

*Main> names p0
["a"]

*Main> eval e0 p0
False

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

How eval works

```
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

  eval e0 (Var "a" :&: Not (Var "a"))
=
  (eval e0 (Var "a")) && (eval e0 (Not (Var "a")))
=
  (lookup e0 "a") && (eval e0 (Not (Var "a")))
=
  True && (eval e0 (Not (Var "a")))
=
  True && (not (eval e0 (Var "a")))
= ... =
  True && False
=
  False
```

Propositions

```
p1 :: Prop
p1 =  (Var "a" :&: Var "b") :|:
      (Not (Var "a") :&: Not (Var "b"))

e1 :: Env
e1 =  [("a",False), ("b",False)]

*Main> showProp p1
((a&b)|((~a)&(~b)))

*Main> names p1
["a","b"]

*Main> eval e1 p1
True

*Main> lookUp e1 "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

```
p1 :: Prop
p1 =  (Var "a" :&: Var "b") :|:
      (Not (Var "a") :&: Not (Var "b"))

*Main> envs (names p1)
[[("a",False),("b",False)],
 [("a",False),("b",True)],
 [("a",True ),("b",False)],
 [("a",True ),("b",True )]]

*Main> [ eval e p1 | e <- envs (names p1) ]
[True,
 False,
 False,
 True]

*Main> satisfiable p1
True
```

Part IX

Aside:

All sublists of a list

All sublists of a list

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

All sublists 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"]]
```

Part X

The Universal Type and Micro-Haskell

The Universal Type and Micro-Haskell

```
data  Univ  =  UBool Bool
            |  UInt Int
            |  UList [Univ]
            |  UFun (Univ -> Univ)

data  Hask  =  HTrue
            |  HFalse
            |  HIf Hask Hask Hask
            |  HLit Int
            |  HEq Hask Hask
            |  HAdd Hask Hask
            |  HVar Name
            |  HLam Name Hask
            |  HApp Hask Hask

type  HEnv  =  [(Name, Univ)]
```

Show and Equality for Universal Type

```
showUniv :: Univ -> String
showUniv (UBool b)    =  show b
showUniv (UInt i)     =  show i
showUniv (UList us)   =
  "[" ++ concat (intersperse "," (map showUniv us)) ++ "]"

eqUniv :: Univ -> Univ -> Bool
eqUniv (UBool b) (UBool c)     =  b == c
eqUniv (UInt i) (UInt j)       =  i == j
eqUniv (UList us) (UList vs)   =
  and [ eqUniv u v | (u,v) <- zip us vs ]
eqUniv u v                     =  False
```

Can’t show functions or test them for equality.

Micro-Haskell in Haskell

```
hEval :: Hask -> HEnv -> Univ
hEval HTrue r           =  UBool True
hEval HFalse r          =  UBool False
hEval (HIf c d e) r     =
  hif (hEval c r) (hEval d r) (hEval e r)
  where  hif (UBool b) v w  =  if b then v else w
hEval (HLit i) r        =  UInt i
hEval (HEq d e) r       =  heq (hEval d r) (hEval e r)
  where  heq (UInt i) (UInt j) = UBool (i == j)
hEval (HAdd d e) r      =  hadd (hEval d r) (hEval e r)
  where  hadd (UInt i) (UInt j) = UInt (i + j)
hEval (HVar x) r        =  lookUp r x
hEval (HLam x e) r      =  UFun (\ v -> hEval e ((x,v):r))
hEval (HApp d e) r      =  happ (hEval d r) (hEval e r)
  where  happ (UFun f) v  =  f v

lookUp :: HEnv -> Name -> Univ
lookUp x r  =  head [ v | (y,v) <- r, x == y ]
```

Test data

```
h0 =
 (HApp
   (HApp
     (HLam "x" (HLam "y" (HAdd (HVar "x") (HVar "y"))))
     (HLit 3))
   (HLit 4))

test_h0 = eqUniv (hEval h0 []) (UInt 7)
```