Informatics 1A

Functional Programming Lecture 9

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 Either a b = Left a | Right b
```

Part II

Boolean

Boolean

```
import Prelude
  hiding (Bool(True, False), (&&), (||), not)

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

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
  deriving (Eq, Show)

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

```
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
  deriving (Eq, Show)

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

```
import Prelude hiding ((++), map, filter)

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

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

map :: (a -> b) -> List a -> List b
map f Nil              =  Nil
map f (Cons x xs)      =  Cons (f x) (map f xs)

filter :: (a -> Bool) -> List a -> List a
filter p Nil                          =  Nil
filter p (Cons x xs) | p x            =  Cons x (filter p xs)
                     | otherwise      =  filter p xs
```

Part VI

Natural numbers

Defining arithmetic by recursion (wrong)

```
import Prelude hiding ((+), (*), (^))

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

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

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

Defining arithmetic by recursion (right)

```
import Prelude hiding ((+), (*), (^))

data  Nat   =   Zero
            |   Succ Nat
  deriving (Eq, Show)

(+) :: Nat -> Nat -> Nat
Zero + n        =   n
(Succ m) + n    =   Succ (m + n)

(*) :: Nat -> Nat -> Nat
Zero * n        =   Zero
(Succ m) * n    =   (m * n) + n

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