Informatics 1

Functional Programming Lecture 9

Algebraic Data Types

Don Sannella

University of Edinburgh

Part I

Algebraic types

Algebraic types are the most important component of functional programming that I haven't covered yet.

We've seen lots of types: integers, floating point numbers, characters, booleans.
Also ways of building types: lists, functions, tuples. All very useful, built in to Haskell.
We get lists of integers, lists of functions from integers to lists of booleans, etc.
An infinite number of types built in a finite number of ways.

Algebraic types is about how to build new types in an INFINITE number of ways.
This is where most of those other types came from - you could define them yourself, if they weren't built in.

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
```

Here are 10 examples, defined completely in 10 lines.
Some you've seen already (Bool). List is [...] and Pair is 2-tuples, both with a different notation.
We'll look at them one at a time. The general case will emerge through the examples.

Part II

Boolean

We'll start with a simple example: Booleans. Where do they come from?
What if I needed them and they weren't already in Haskell?

Boolean

```
data Bool = False | True
```

Bool: name of new type, needs to begin with uppercase letter.
Constructors False and True, need to begin with uppercase letter.
As many as you want, separated by a vertical bar.

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

False and True are the only values of Bool, and they are different.
Then we can define new functions on Bool using pattern matching.

These definitions are essentially the truth tables.

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

True is the identify for &&

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

False is the identify for ||

These are just like the definitions you've been writing for functions on lists.
The difference is that we've defined the type ourselves, and patterns use the constructors in the type definition.

Boolean — eq and show

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

Here's a definition of what it means for two Bool values to be equal.
Four cases - just write them out.

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

:: Bool :: String

Defines how to display Bool values by converting them to String.

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
```

Bool had two constructors, Season has four.
Values are the four seasons.

Function next tells you which Season comes next in the year.

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
```

Equality on Season - writing all combinations requires 16 cases.
(No, you can't use repeated variables to abbreviate the first 4 cases - not allowed in patterns.)

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

Converting Season to printable values.

There is a way to get Haskell to define these functions automatically - coming later ("type classes").
There is also a way to get Haskell to incorporate these functions into the built-in == and show functions.

Seasons and integers

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

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

These functions convert back and forth from Seasons to Int.
Notice, Seasons aren't REPRESENTED by Ints.
The constructors (Winter etc.) ARE the values.
No other representation is required.

```
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)
```

Then we can give a simpler definition of next.

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

Ditto for equality.

Part IV

Shape

Bool and Season were defined by enumerating their values, represented by constructors.
Shape is different - its constructors take values of another type as arguments.

Shape

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

Here we define type SYNONYMS - a new name for an old type.
Just to help us remember what these numbers mean.
3.1 :: Float, also 3.1 :: Radius etc.

```
data   Shape   =   Circle Radius
               |   Rect Width Height
```

A Shape is either a Circle with a radius,
or a Rect (rectangle) with a width and height.

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

These constructors take arguments:
Circle takes one argument, Rect takes two.
The type definition gives the argument types.

We define the area of a shape by giving the cases for circles and rectangles separately, using patterns.
This uses constructors for distinguishing between cases, variables for extracting values from data.

Circle :: Radius -> Shape and Rect :: Width -> Height -> Shape are functions.

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 ++ ")"
```

Definitions of equality and show function on values of type Shape.
The show function uses a helper function to put parentheses around negative numbers.

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
```

Patterns with variables make it possible to write function definitions very concisely.

We can do without patterns if we define these functions. isCircle and isRect for testing which kind of Shape, radius, width and height for extracting values from Shapes.

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"
```

Here is how we would have to write the area function if we use those test and extraction functions instead of patterns. Yuck!

This is the way the computer executes our 2-line definition earlier.

Part V

Lists

Lists

List is a PARAMETRISED type - a type-level function, that takes a type as argument.
This gives us types that depend on other types.

With declarations

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

A value of type List a is either Nil (empty)
or Cons followed by a value of type a
and a value of type List a.
Note: RECURSION

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

Now we can define append, and other functions on List.
Using recursion, just like the type definition uses recursion.

With built-in notation

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

Here's the same thing, using Haskell's built-in list notation.
List a = [a], Nil = []. Cons a l = a:l

Part VI

Natural numbers

Naturals

With names

```
data  Nat  =  Zero
           |  Succ Nat
```

Natural numbers (0, 1, 2, ...).
Recursive, like List, but not parametrised.

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

nth power of a Float.

With built-in notation

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

Numbers in Haskell aren't defined this way! Imagine writing 1000000 as succ(...(succ Zero)...)
Haskell uses ordinary computer arithmetic.

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
```

We can define addition and multiplication in the same style.

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
```

Here's what the same definitions would look like, using Haskell's normal arithmetic notation.