Algebraic Datatypes in Haskell

CS 430 :: Bowers :: Spring '19

An **algebraic data type** is a type formed by combining other types.

- Sort of like a `struct` in C, only *better* because it enables pattern matching.
- Defines what values the type can take:

```
data Bool = False | True
data Int = -2147483648 | -2147483647 | ... | -1 |
           0 | 1 | 2 | ... | 2147483647

data Boolish = False | True | OutcomeInDoubt
data Year = Year Int
data Maybe a = Just a | Empty

tenYearsFromNow :: Year -> Year
tenYearsFromNow (Year x) = (Year (x + 10))
twentyYearsFrom y = tenYearsFromNow $ tenYearsFromNow y

tyfnAsInt :: Year -> Int
tyfnAsInt (Year x) = (x + 10)
```

(Think of | as *or*.)

Think of a type as a set of values (forget the operations for now).

i.e. $\mathbb{Z}$ is the integers.

We can take **products** of sets:

- $\mathbb{Z} \times \mathbb{R}$ is all ordered pairs (x, y) where $x \in \mathbb{Z}$ and $y \in \mathbb{R}$.

We can take **unions** of sets:

- $\{1, 2, 3\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$.
- Sometimes written $\{1, 2, 3\} + \{2, 3, 4\} = \{1, 2, 3, 4\}$

So we have operations $\times$ and $+$ on sets (looks like algebra), and this in turn is easily extended to types.

Products of Types

```haskell
data Point = Point Float Float
```

If $Float$ is the set of all floats, then `Point Float Float` is like saying $Point = Float \times Float$.

We can now create a point like `Point 3.4 4.7`.

The word `Point` is called a **value constructor** and the `Point Float Float` above says that a `Point` holds two `Float` values.

To create a point, you just specify the value constructor and two floats:

```haskell
Point 3.0 4.8
```

Similarly, we can sum types.

```haskell
data Point -- A point is either 2D or 3D
 = Point2D Float Float
 | Point3D Float Float Float
```

```haskell
data Point = Point2D Float Float
 | Point3D Float Float Float

data Segment = Segment (Point, Point)
```

Algebraic datatypes are especially powerful when mixed with pattern matching:

```haskell


xCoord :: Point -> Float
xCoord (Point2D x _)   = x
xCoord (Point3D x _ _) = x

yCoord :: Point -> Float
yCoord (Point2D _ y)   = y
yCoord (Point3D _ y _) = y

zCoord :: Point -> Float
zCoord (Point3D _ _ z) = z

start (Segment (p1, _)) = p1
end (Segment (_, p2)) = p2

xCoord $ start (Segment ((Point2D 3 4), (Point2D 5 6)))
```

Consider the following C style struct for a tree and suppose we only want to store data at the leaf nodes:

```c
typedef struct node {
    struct node *left;
    struct node *right;
    int value; // Storing values at internal nodes also
} node;


bool isLeaf(struct node *n) {
  // Wouldn't it be nice if leaf nodes looked different
  // than internal nodes?
  return (*n).left == NULL && (*n).right == NULL;
}
```

Haskell version:

```haskell
data Tree
  = Empty
  | Leaf Int
  | Node Tree Tree
```

Read this as: "a `Tree` is either the `Empty` tree, or is a `Leaf` node storing a value, or is a `Node` with two children, both of which are `Tree` typed."

Notice that the definition of `Tree` uses `Tree` *within the definition*. This is a **recursive type**.

Haskell version:

```haskell
data Tree
  = Empty
  | Leaf Int
  | Node Tree Tree
```

How would you code for the following tree?

```
     .
    / \
   .   3
  / \
 4   5
```

Haskell version:

```haskell
data Tree
  = Empty
  | Leaf Int
  | Node Tree Tree
```

How would you code for the following tree?

```
     .
    / \
   .   3
  / \
 4   5
```

```haskell
Node (Node (Leaf 4) (Leaf 5)) (Leaf 3)
```

```haskell
data Tree
  = Empty
  | Leaf Int
  | Node Tree Tree
```

Allows for some great pattern matching:

```haskell
allValues :: Tree -> [Int]
allValues t = case t of
  Empty -> []
  (Leaf x) -> [x]
  (Node l r) -> (allValues l) ++ (allValues r)
```

```
> allValues (Node (Node (Leaf 4) (Leaf 5)) (Leaf 3))
```

```haskell
data Tree
  = Empty
  | Leaf Int
  | Node Tree Tree
```

Something interesting:

```
*Main> :t Node
Node :: Tree -> Tree -> Tree
*Main> :t Leaf
Leaf :: Int -> Tree
```

So even *value constructors* are really *functions*!

In GHCI we'd like to print out the value of a tree, like:

```
> (Node (Leaf 2) (Leaf 3))
```

but this will give us an error. Haskell only prints things that are part of the **type class** `Show`. Fortunately, we can get this almost automagically:

```
data Tree
  = Empty
  | Leaf Int
  | Node Tree Tree
  deriving (Show)
```

This is sort of like `implements Show` in Java. Just like algebraic types are sort of like `structs`, type classes are sort of like interfaces. Now our `Tree` is "showable".

Other typeclasses you might want: `Eq` (can test equality), `Ord` (can order / compare), `Show` (can convert to a string), `Read` (can read from a string).

```haskell
data List a
  = Empty
  | Prepend a (List a)
  deriving (Show, Read, Eq, Ord)

3 `Prepend` (4 `Prepend` (5 `Prepend` Empty))

-- same as

Prepend 3 (Prepend 4 (Prepend 5 Empty))

-- compare to
3 : (4 : (5 : [])) -- i.e. [3, 4, 5]

-- same as
(:) 3 ((:) 4 ((:) 5 []))

listHead (Prepend x _) = x
listTail (_ `Prepend` t) = t

listLength Empty = 0
listLength (x `Prepend` xs) = 1 + (listLength xs)
```

```haskell
data List a
  = Empty
  | Cons a (List a)
  deriving (Show, Read, Eq, Ord)

3 `Cons` (4 `Cons` (5 `Cons` Empty))

-- compare to
3 : (4 : (5 : [])) -- i.e. [3, 4, 5]

listHead (x `Cons` _) = x
-- head (x:_) = x
listTail (_ `Cons` t) = t
-- tail (_:t) = t

listLength Empty = 0
-- length [] = 0
listLength (x `Cons` xs) = 1 + (listLength xs)
--length (x:xs) = 1 + (length xs)
```