Algebraic datatypes

Contents

Define your own type

(Thompson: §5.1, §5.3)

Recall **type synonyms**

```
type Words = String
```

keyword   new name   existing type

What if we want to define our own elements?

```
data Singleton = LonelyElement
```

keyword   name of type   element

This defines a type called `Singleton` with one element, called `LonelyElement`

Define your own type


```haskell
data Singleton = LonelyElement

-- We can define a program
lonelyToZero :: Singleton -> Int
lonelyToZero LonelyElement = 0

-- And the other way round
allToLonely :: Int -> Singleton
allToLonely x = LonelyElement
```

Warnings and errors

- If you run ghci with warnings on (-W) you get a warning. Why?
- If you test your code it gives an error. Why?

Finite sum types

(THOMPSON: §5.3)

What if we want more than on element?

```
data Singleton = LonelyElement | AnotherElement
```

keyword name of type element second element

Analogy with sets

- We can build any finite set from singletons using the sum $+$
- We can do the same in Haskell using `|`
- It is called a **sum type**

Finite sum types

(Thompson: §5.3)

Example This is how `Bool` is defined in Haskell (which is built-in)

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

Finite sum types: colours

Example We can define more than two elements

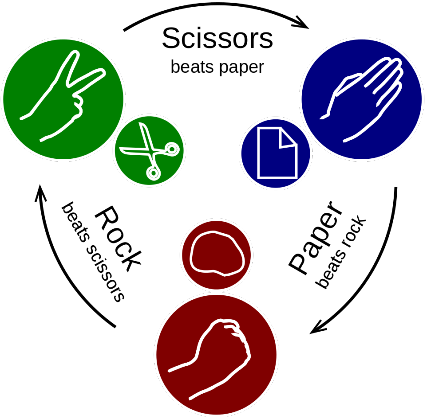
```
  data Colour = Red | Green | Blue | Indigo | Violet
    deriving Show
```

Finite sum types: Rock paper scissors

(Thompson: §5.3)

Example We can define more than two elements

```haskell
data Colour = Rock | Paper | Scissors  deriving Show
```



Sum types

With sets, we can use sums to define $\mathbb{Z} + \mathbb{B}$ or $\mathbb{Z} + \mathbb{Z} + \mathbb{B}$

We can do the same in Haskell with `|`, we just need to choose tag names

```haskell
data IntPlusBool = First Int | Second Bool  deriving Show
```

keyword

name of type

tag or constructor

argument

```haskell
data IntPlusIntPlusBool =
  First Int | Second Int | Third Bool  deriving Show
```

Product types

(Thompson: §5.3)

With sets, we can use products to define $\mathbb{Z} \times \mathbb{B}$, which has elements (n, bool)

Suppose we want to make a **product type** of `Int` and `Bool`

Method 1: using brackets

```
type IntAndBool = (Int, Bool)
```

Method 2: using a constructor

```
data IntAndBool' = Combine Int Bool
```

Difference

- Method 1 has elements of the form `(0, True)` and `(9, False)`
- Method 2 has elements of the form `Combine 0 True` and `Combine 9 False`

Product types

```
-- Method 1
type IntAndBool = (Int, Bool)

fst :: (Int, Bool) -> Int        -- this is built-in
fst (n, _) = n

-- Method 2
data IntAndBool' = Combine Int Bool

fst' :: IntAndBool' -> Int       -- how can we define this?

-- how about a function like this:
pairing :: Int -> Bool -> IntAndBool'
```

Product types

Product types can have more than two arguments

```haskell
-- Method 1
type IntAndIntAndBool = (Int, Int, Bool)

-- Method 2
data IntAndIntAndBool' = Combine Int Int Bool
```

Combining products and sums

(Thompson: §5.3)

```haskell
type Name = String                -- type synonym for String

data Accommodation =
  Hotel Name Double Double
  | Hostel Name Int Double
  | Airbnb Name Double
  | Friend Name
```

Maybe

(Thompson: §14.4, §14.6)

The `Maybe` datatype is defined as follows

```
data Maybe a = Just a | Nothing
```

Example For `a = Int` this gives

```
data Maybe Int = Just Int | Nothing
```

We use this if we are unsure we have an integer or not

Summary

Sum types (constructors can have zero arguments)

```
data IntPlusBool = First Int | Second Bool  deriving Show
```

keyword   name of type   tag or constructor   argument

Product types

```
type IntAndBool = (Int, Bool)
data IntAndBool' = Combine Int Bool
```

keyword   name of type   constructor   arguments

Sum types and products types can be combined

Next week

- Cases
- Lists
- Recursion