COMP105 Lecture 20

Custom Types

Outline

Today

- Creating our own types
  - The `type` keyword
  - The `data` keyword
  - Records

Relevant book chapters

- Programming In Haskell Chapter 8
- Learn You a Haskell Chapter 8

The `type` keyword

The **type** keyword gives a new name to an existing type

- All types must start with capital letters

```haskell
type String' = [Char]
```

```haskell
exclaim :: String' -> String'
exclaim str = str ++ "!"
```

```
ghci> exclaim "hello"
"hello!"
```

The type keyword

type is useful when you want to give a **meaningful name** to a complex type

```
type VoteResults = [(Int, String)]
```

```
results :: VoteResults
results = [(2, "red"), (1, "blue"), (1, "green")]
```

```
ghci> head results
(2,"red")
```

The data keyword

The **data** keyword is used to create an entirely new type

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

- | should be read as “or”
- each of the values is a **constructor**

The data keyword

```
data Direction = North | South | East | West
```

```
rotate North = East
rotate East = South
rotate South = West
rotate West = North
```

```
ghci> :t rotate
rotate :: Direction -> Direction
```

Type classes

By default, a new data type is **not** part of any type class

```
ghci> rotate North
No instance for (Show Direction) arising from ...
```

Type classes

We can use the **deriving** keyword to fix this

```
data Direction = North | South | East | West
                                  deriving (Show)
```

```
ghci> rotate North
East
```

Haskell automatically writes the `show` function for us

- You can override this if you want

Type classes

Haskell can **automatically** implement the following type classes

- `Show` – will print out the type as it is in the code
- `Read` – will parse the type as it is in the code
- `Eq` – the natural definition of equality
- `Ord` – constructors that come first are smaller

Exercise

Consider the following type:

```
data Colour = Red | Blue | Green
                                deriving (Show, Read, Eq, Ord)
```

What are the results of the following queries?

1. `show Red ++ show Blue ++ show Green`
2. `Red == Red && Red /= Green`
3. `Red < Blue && Green < Blue`
4. `read "red" :: Colour`

More complex constructors

More complex constructors can contain **other types**

```
data Point = Point Int Int deriving (Show, Read, Eq)
```

```
ghci> Point 1 4
Point 1 4

ghci> read "Point 10 10" :: Point
Point 10 10

ghci> Point 2 2 /= Point 3 1
True
```

More complex constructors

It is common to use **pattern matching** to work with complex constructors

```
shift_up (Point x y) = Point x (y+1)
```

```
ghci> shift_up (Point 1 1)
Point 1 2
```

```
ghci> :t shift_up
shift_up :: Point -> Point
```

Example

```
move :: Point -> Direction -> Point

move (Point x y) North = Point x (y+1)
move (Point x y) South = Point x (y-1)
move (Point x y) East  = Point (x+1) y
move (Point x y) West  = Point (x-1) y
```

```
ghci> move (Point 0 0) North
Point 0 1
```

Even more complex constructors

Types can have **multiple constructors** each of which can have their own types

```
data Shape = Circle Float | Rect Float Float
                                      deriving (Show)
```

```
ghci> :t Circle 2.0
Circle 2.0 :: Shape
```

```
ghci> :t Rect 3.0 4.0
Rect 3.0 4.0 :: Shape
```

Example

```haskell
area :: Shape -> Float

area (Circle radius) = pi * radius**2
area (Rect x y) = x * y
```

```
ghci> area (Circle 2.0)
12.566371

ghci> area (Rect 3.0 4.0)
12.0
```

Records

You can use data types to build custom **records**...

```
data Person = Person String String Int String
```

```
get_first_name  (Person x _ _ _) = x
get_second_name (Person _ x _ _) = x
get_age         (Person _ _ x _) = x
get_nationality (Person _ _ _ x) = x
```

```
ghci> get_age (Person "joe" "bloggs" 25 "UK")
25
```

Record syntax

To make things easier, Haskell provides a **record syntax**

```
data Person = Person { firstName :: String,
                       secondName :: String,
                       age :: Int,
                       nationality :: String}
                                       deriving(Show)
```

```
ghci> Person "joe" "bloggs" 25 "UK"
Person {firstName = "joe", secondName = "bloggs",
          age = 25, nationality = "UK"}
```

Record syntax

When you use the record syntax, Haskell automatically creates **getter** functions for each parameter

```
gchi> let joe = Person "joe" "bloggs" 25 "UK"
```

```
gchi> firstName joe
"joe"
```

```
ghci> secondName joe
"bloggs"
```

Record syntax

Records can be created **out of order** (normal data types cannot)

```
data Example = Example { a :: String, b :: Int}
                                  deriving (Show)
```

```
ghci> Example "one" 2
Example {a = "one", b = 2}
```

```
ghci> Example {b = 3, a = "zero"}
Example {a = "zero", b = 3}
```

Example

```
data AdvShape =   AdvCircle Point Float
                | AdvRect   Point Point
                              deriving (Show)


area' (AdvCircle _ radius) = pi * radius**2


area' (AdvRect (Point x1 y1) (Point x2 y2)) =
    let
        w = abs (x1 - x2)
        h = abs (y1 - y2)
    in
        fromIntegral (w * h)
```

Exercises

1. Use the type syntax to create a type ListListInt that is the same as a list of lists of integers.
2. Using the Direction type, write a function flip that returns the opposite direction to the input (so flip North will return South).
3. Using the Point type, write a function add_points that adds two points together.
4. Create a type ThreeDPoint with one constructor that has three integers.

Exercises

5. Create a type `ThreeDObject` with two constructors `Sphere` (one `Float` for the radius) and `Cube` (three `Floats`: height, width, and depth)

6. Write a function `volume` that takes a `ThreeDObject` and returns the volume of that object.

7. Use the record syntax to create a type `Module` that has parameters for the module code, module title, lecturer, and number of students.

Summary

- Creating our own types
  - The `type` keyword
  - The `data` keyword
  - Records

Next time: Parameterized custom types