Introduction To Haskell Programming

Prof. S.P Suresh

Chennai Mathematical Institute

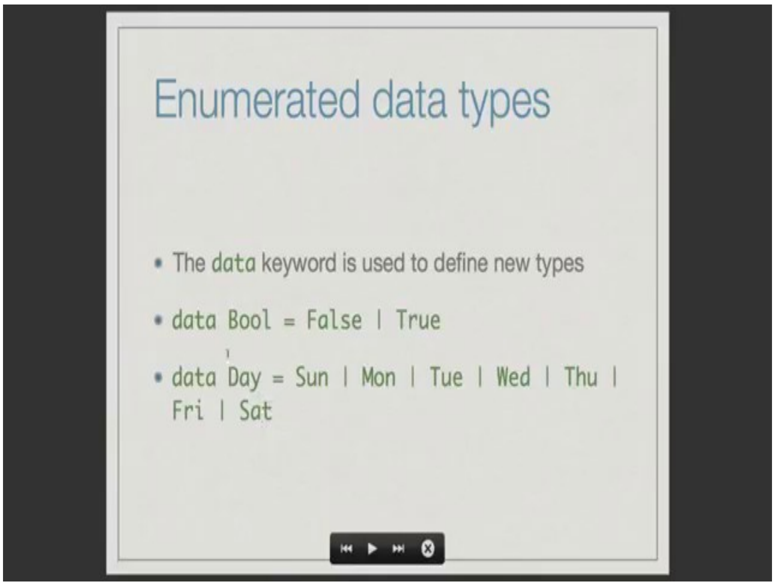
Module # 05

Lecture - 01

User Defined Data Types

Welcome to week 5 of the NPTEL course on Functional Programming in Haskell.

(Refer Slide Time: 00:19)



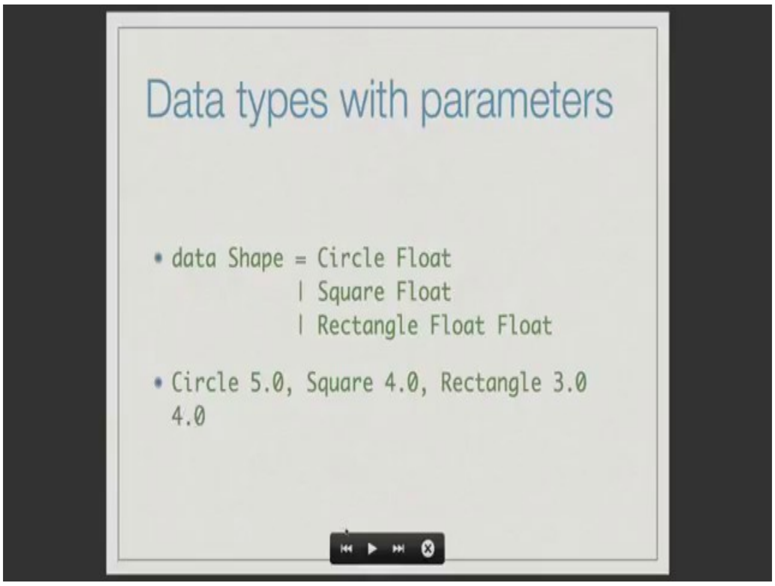
Enumerated data types

- The data keyword is used to define new types
- data Bool = False | True
- data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

I am S.P. Suresh and I shall be taking over from Professor Madhavan Mukund for the next few weeks. In today's lecture, we shall be looking at user defined data types in Haskell. The simplest way to define new data types in Haskell is the so called enumerated data types, here is an example data Bool = False | True. You define the new data type using the data keyword and give the name of the type here Bool.

Notice the capitalization and then, you enumerate all the values that will be part of the type, in this case it is either False or True, notice the capitalization again. Another example is Day, you declare it by saying data Day = Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday. These are all the possible values of the type Day.

(Refer Slide Time: 01:15)

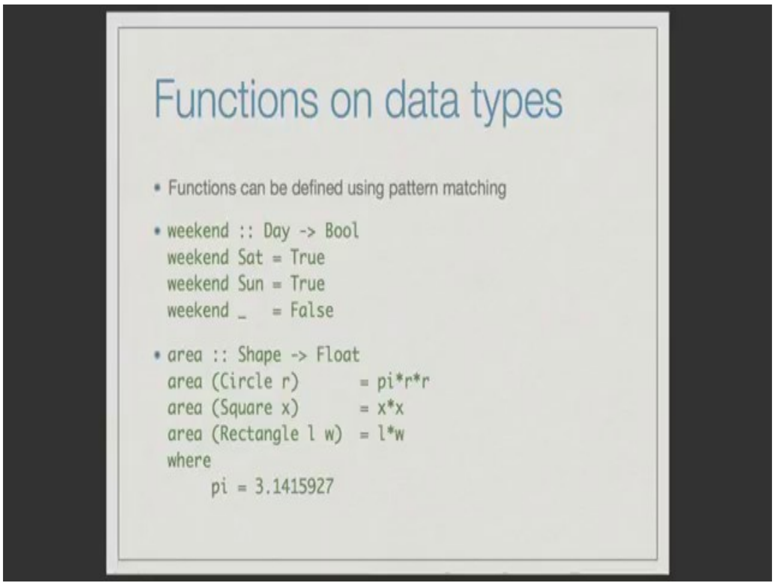


Data types with parameters

- data Shape = Circle Float
  ```
  data Shape = Circle Float
             | Square Float
             | Rectangle Float Float
  ```
- Circle 5.0, Square 4.0, Rectangle 3.0 4.0

Here is another example, data Shape equals either a Circle or a Square or a Rectangle, but a Circle has a parameter namely the radius and a Square comes with the parameter, namely the length of the side. Similarly, a Rectangle also has a parameter namely the length and breadth. This enables us to declare new data types, which can take infinitely many values, you have Circle 1, Circle 2.0 or Circle 3.0 and so on, one Circle object for every float value. Here are some examples, Circle 5.0, Square 4.0, Rectangle 3.0, 4.0; this is a Rectangle with breadth 3 and length 4.

(Refer Slide Time: 02:15)



Functions on data types

- Functions can be defined using pattern matching

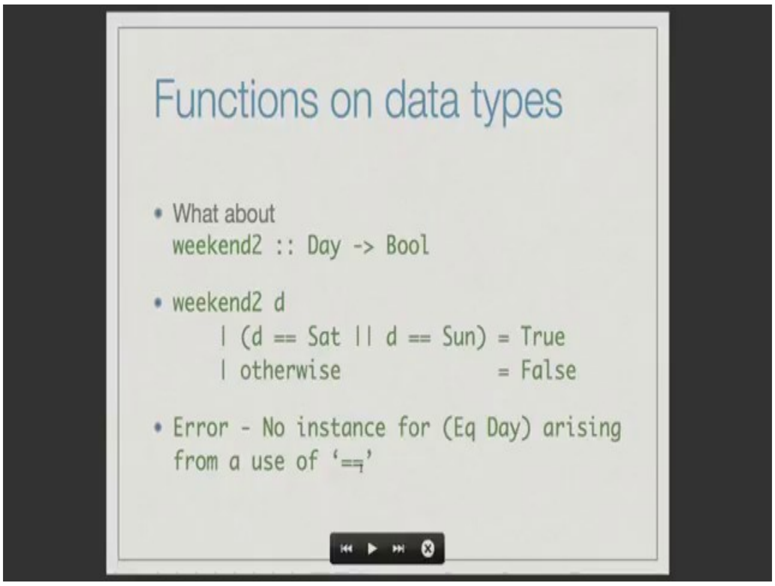
-
  ```
  weekend :: Day -> Bool
  weekend Sat = True
  weekend Sun = True
  weekend _   = False
  ```

-
  ```
  area :: Shape -> Float
  area (Circle r)        = pi*r*r
  area (Square x)        = x*x
  area (Rectangle l w)   = l*w
  where
       pi = 3.1415927
  ```

You can define functions on user defined data types in the usual manner, here are some example functions defined using pattern matching. For instance, you can define what a weekend is. weekend is a function from Day -> Bool and here is the definition, weekend of

Saturday is True, weekend of Sunday is True and weekend of anything else is False. Similarly, you can define an area function, which is a function from Shape to Float, area of a Circle with radius r is pi*r*r, area of a Square with length of side x is x*x and area of a Rectangle with length l and width w is l*w. Are there other ways to define functions on user defined data types, yes.

(Refer Slide Time: 03:11)



Functions on data types

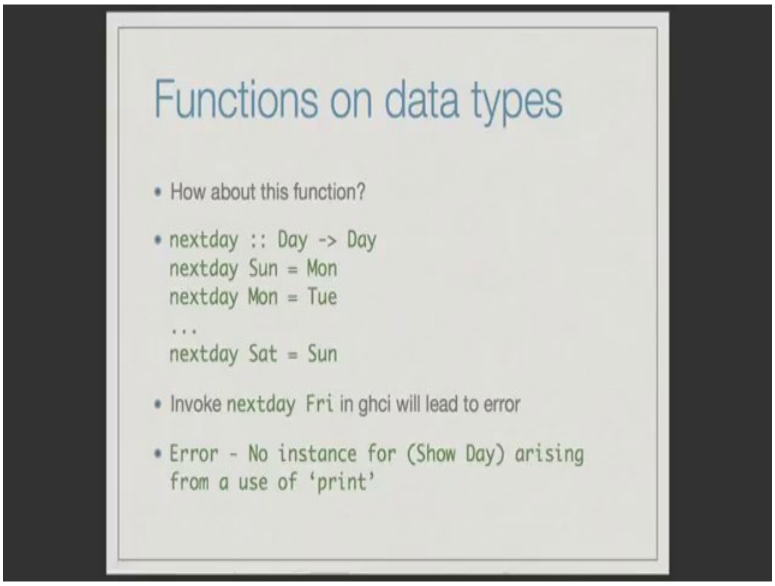
- What about
  `weekend2 :: Day -> Bool`
- weekend2 d

```
weekend2 d
     | (d == Sat || d == Sun) = True
     | otherwise              = False
```

- Error - No instance for (Eq Day) arising from a use of '=='

For instance, here is another way to define the weekend function called weekend2, the definition is here. Weekend2 of d is True if d is equal to Saturday or d is equal to Sunday and it is equal to False otherwise. But, if you enter this program as it is in ghci, you will get an error, the error message will be something like this. No instance for (Eq Day) arising from a use of the equality operator; let us see how to fix this later.

(Refer Slide Time: 03:49)



Functions on data types

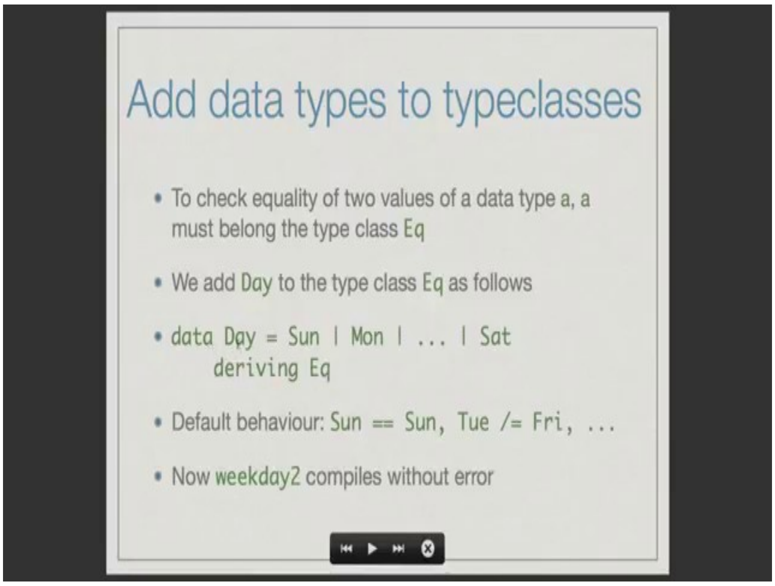
- How about this function?
- nextday :: Day -> Day

```
nextday :: Day -> Day
nextday Sun = Mon
nextday Mon = Tue
...
nextday Sat = Sun
```

- Invoke nextday Fri in ghci will lead to error
- Error - No instance for (Show Day) arising from a use of 'print'

Here is another function you can write, which is a function from Day -> Day, it gives the nextday. For instance, nextday Sun = Mon, nextday Mon = Tue and so on, nextday Sat = Sun. Now, if you load this function in ghci and invoke nextday Fri, it will again lead to an error and it is the following error. No instance for (Show Day) arising from a use of print, we will see what this means and how to fix this in a later slide.

(Refer Slide Time: 04:26)



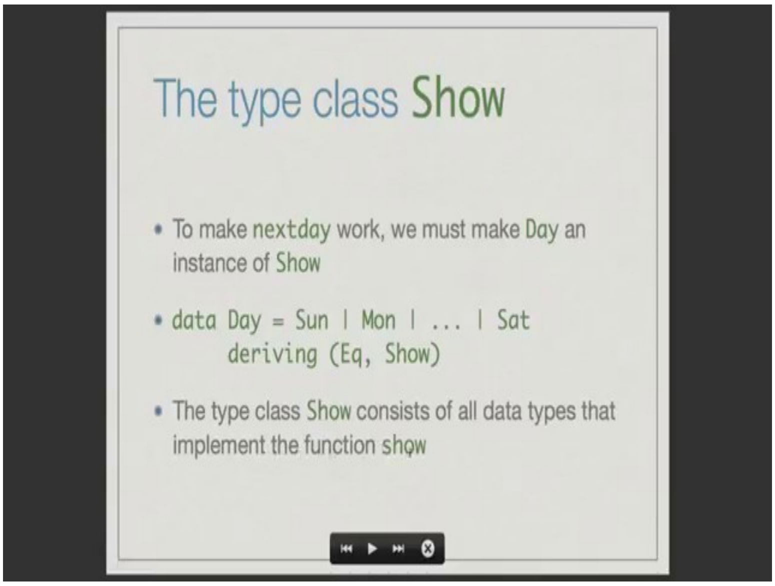
Add data types to typeclasses

- To check equality of two values of a data type a, a must belong the type class Eq
- We add Day to the type class Eq as follows
- data Day = Sun | Mon | ... | Sat
  deriving Eq
- Default behaviour: Sun == Sun, Tue /= Fri, ...
- Now weekday2 compiles without error

To check equality of two values of a data type a, a must be declared to belong to the type class Eq. We have seen the notion of a type class in earlier lectures and we have also seen the type class Eq in detail. So, to get weekday2 to work we need to add Day to the type class Eq and we add Day to the type class Eq as follows, data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat deriving Eq, the keyword is 'deriving'.

When we declare Day to derive from equality, the equality operator has the default behavior. Sunday ==Sunday, Tuesday /= Friday, Monday == Monday, etcetera. Once you declare Day to be deriving from Eq, then weekday2 compiles without error.

(Refer Slide Time: 05:32)

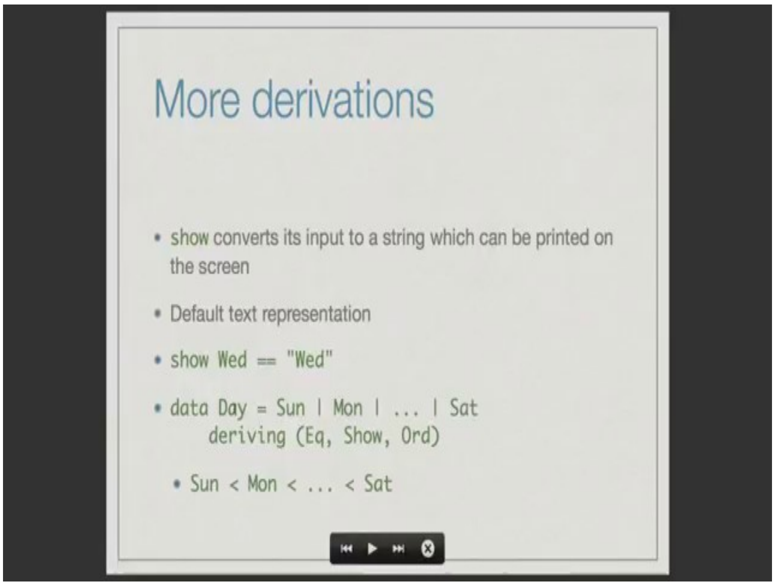


The type class Show

- To make nextday work, we must make Day an instance of Show
- data Day = Sun | Mon | ... | Sat
  deriving (Eq, Show)
- The type class Show consists of all data types that implement the function show

To make the nextday function work, we must make Day an instance of the type class called Show with a capital S. We have to declare it as follows, data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat deriving (Eq, Show). The type class Show consists of all data types that implement the function show with a small s.

(Refer Slide Time: 05:59)



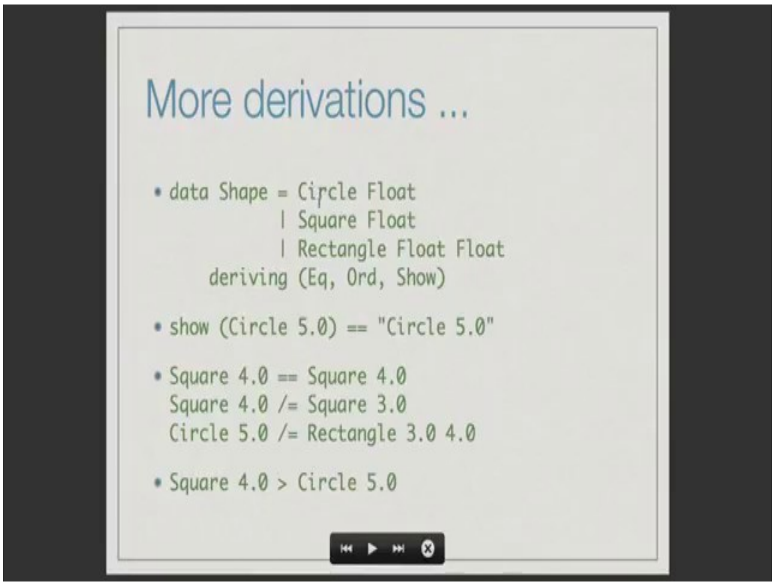
More derivations

- show converts its input to a string which can be printed on the screen
- Default text representation
- show Wed == "Wed"
- data Day = Sun | Mon | ... | Sat
  deriving (Eq, Show, Ord)
  - Sun < Mon < ... < Sat

The function show converts its input to a string which can be printed on the screen, we need not define it explicitly for the data type Day. If we do not give an explicit definition, there is a default definition that Haskell provides, which is just to give a default text representation for the data value. For instance, show Wed is just the string “Wed”, we can also derive Day as an instance of the type class O r d, Ord which is an ordinal type.

When we do this, an order is defined on the data type Day as follows, Sun < Mon < … < Sat and this order is determined by the order in which the data values are enumerated in the data type declaration.

(Refer Slide Time: 07:01)



More derivations ...

- data Shape = Circle Float
                | Square Float
                | Rectangle Float Float
      deriving (Eq, Ord, Show)

- show (Circle 5.0) == "Circle 5.0"

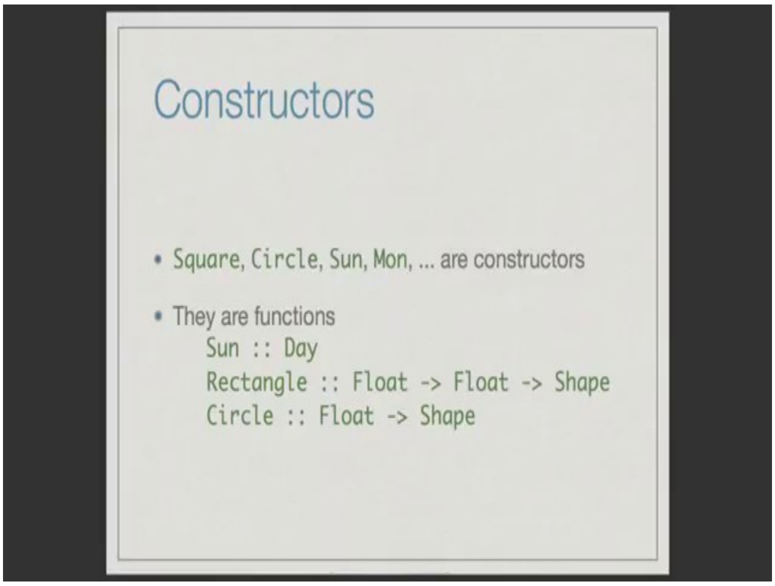
- Square 4.0 == Square 4.0
  Square 4.0 /= Square 3.0
  Circle 5.0 /= Rectangle 3.0 4.0

- Square 4.0 > Circle 5.0

We can also derive Shape to belong to various type classes, we say data Shape = Circle Float | Square Float | Rectangle Float Float deriving (Eq, Ord, Show). Now, you can use all these functions on data on values of type Shape. For instance, Show Circle 5.0 will just be the string that say Circle 5.0, Square 4.0 == Square 4.0, the equality check is derived from the equality on floating point numbers as well as the names, Square or Circle or Rectangle.

For instance, Square 4.0 == Square 4.0, because both the parameters and the name here are equal, Square 4.0 /= Square 3.0. Because, even though the names are equal the parameters are different, Circle 5.0 /= Rectangle 3.0, 4.0, because there are two different types of shape, one is a Circle and other is a Rectangle and we have also derived it to belong to the type class Ord. So, there is an order defined on shapes, Square 4.0 is for instance greater than Circle 5.0, because Square comes later in the declaration than Circle.

(Refer Slide Time: 08:37)



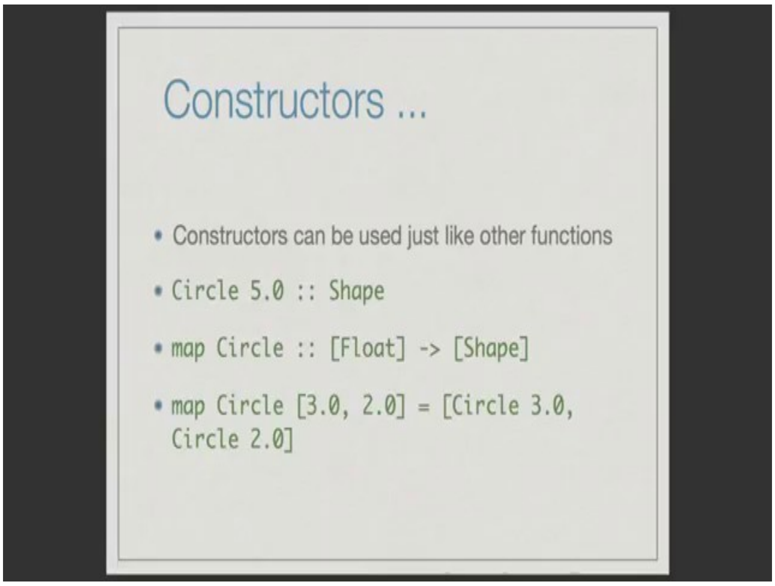
Constructors

- Square, Circle, Sun, Mon, ... are constructors
- They are functions

```
Sun :: Day
Rectangle :: Float -> Float -> Shape
Circle :: Float -> Shape
```

The names Square, Circle, Sun, Mon, etcetera that we have used are called constructors. They are nothing but, functions; Sunday for instance is a function that is of type Day, Sun:: Day. It is a function that accepts no input, but produces an output. Rectangle is a function with two parameters, so it takes two floats as inputs and it produces a shape as output. So, Rectangle is a function whose type is Float -> Float ->Shape, similarly Circle is a function whose type is Float -> Shape.

(Refer Slide Time: 09:21)

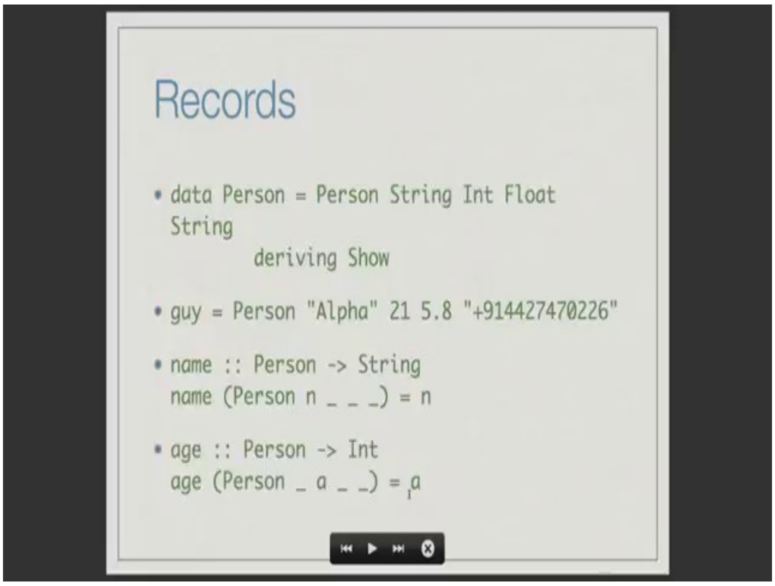


Constructors ...

- Constructors can be used just like other functions
- Circle 5.0 :: Shape
- map Circle :: [Float] -> [Shape]
- map Circle [3.0, 2.0] = [Circle 3.0, Circle 2.0]

These constructors can be used just like any other function, for instance Circle can be invoked on the input 5.0 to give a Shape. You can map the Circle function over a list of Floats to get a list of Shapes. For instance, map Circle on the list [3.0,2.0] which will give you the list consisting of [Circle 3.0, Circle 2.0].

(Refer Slide Time: 09:49)



Records

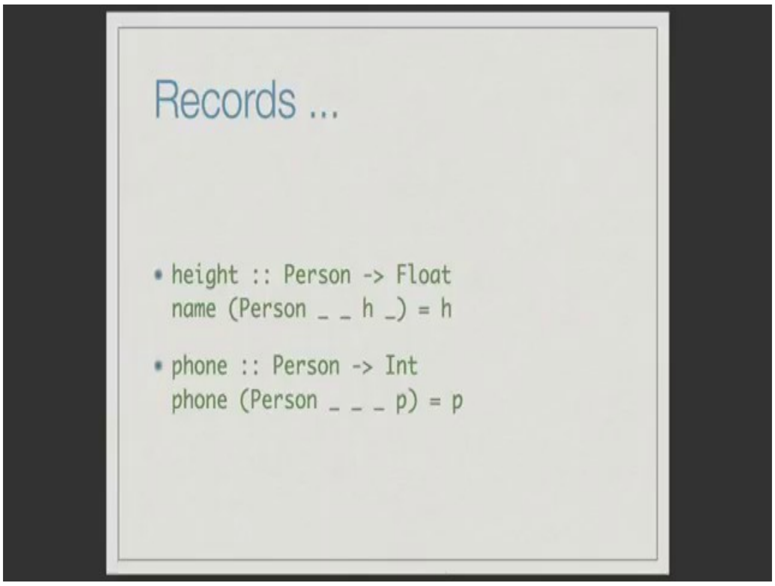
- data Person = Person String Int Float String
  deriving Show
- guy = Person "Alpha" 21 5.8 "+914427470226"
- name :: Person -> String
  name (Person n _ _ _) = n
- age :: Person -> Int
  age (Person _ a _ _) = a

Here is another way of defining types, data Person = Person String Int Float String. Now, you will see two occurrences of the word Person here, one on the left one on the right. On the left, it denotes the name of the type; on the right it is the name of the constructor. Now, unlike in a type like Shape where we had three constructors Circle, Square, Rectangle, here we have only one constructor, though we have many parameters.

The convention in Haskell is that, if a data type has only one constructor then you use the same name for both the type and the constructor. So, the Person that appears on the right is a constructor and the Person that appears on the left is the name of the data type. So, here we say that data Person = Person String Int Float String, the intention is that the first string is the name of the person, this int here is the age of the person, the float here is let us say the height of the person and the last string here is the phone number.

So, for instance I might say guy = Person “Alpha” 21, 5.8 and some phone number. How do you extract the name of a Person object? You write a function name :: Person -> String, whose input is Person and the output is String and the definition is this. Name, person n, any age, any height and any phone number equals n written as name (Person n _ _ _) = n, here is a function that extracts the age of a person, age:: Person-> Int, it is a function from Person to Int and the definition is age (Person _ a _ _) = a.

(Refer Slide Time: 11:45)



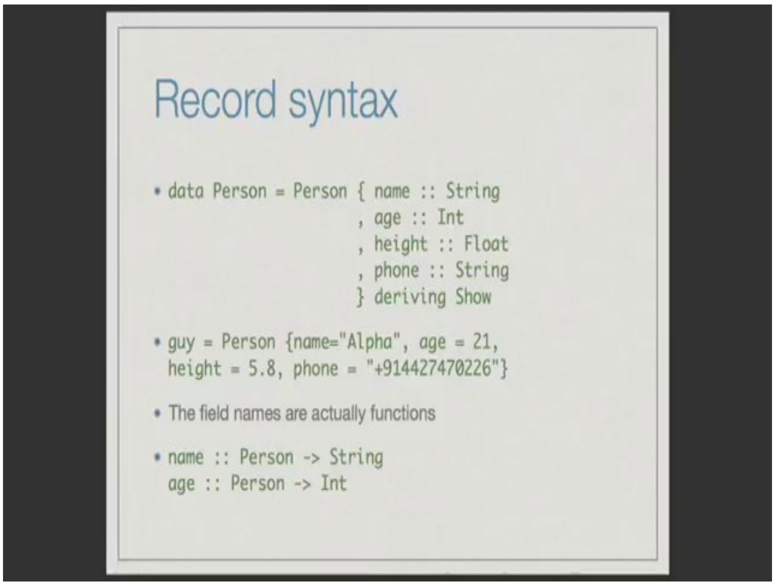
Records ...

-
  ```
  height :: Person -> Float
  name (Person _ _ h _) = h
  ```

-
  ```
  phone :: Person -> Int
  phone (Person _ _ _ p) = p
  ```

You can write a height function which says height (Persin _ _ h _) = h and here is a function that accepts the phone number of a person; phone (Person _ _ _ p) = p. There is a pattern matching and there is a don’t care pattern in all these definitions, but this kind of definition is quite cumbersome. So, Haskell offers an alternative easier syntax which is also familiar from other languages.

(Refer Slide Time: 12:21)



Record syntax

-
  ```
  data Person = Person { name :: String
                       , age :: Int
                       , height :: Float
                       , phone :: String
                       } deriving Show
  ```

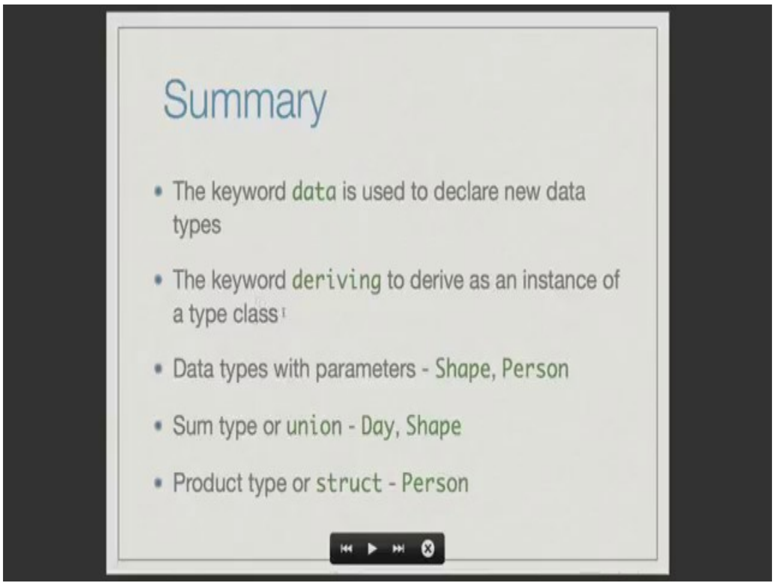
-
  ```
  guy = Person {name="Alpha", age = 21,
  height = 5.8, phone = "+914427470226"}
  ```
- The field names are actually functions

-
  ```
  name :: Person -> String
  age :: Person -> Int
  ```

You can define the Person type as for this : data Person = Person {name::String, age :: Int, height :: Float, phone :: String} deriving Show. data Person equals Person and within braces you say name which is of type String, age which is of type Int, height which is of type Float and phone which is of type String. So, you are naming all the fields directly in the data definition itself and these names here, name, age, height, phone etc are really nothing but, the functions that we defined earlier. Name is a function from Person -> String, age is a function from Person -> Int.

You declare new objects of type person in this syntax as follows, guy = Person {name = “Alpha”, age= 21, height =5.8, phone =”+914427470226”} guy equals Person, within braces you say name equals alpha, age equals 21, height equals 5.8 and phone equals phone number. This is an easier syntax that Haskell offers.

(Refer Slide Time: 13:24)



Summary

- The keyword data is used to declare new data types
- The keyword deriving to derive as an instance of a type class
- Data types with parameters - Shape, Person
- Sum type or union - Day, Shape
- Product type or struct - Person

To summarize, you have seen simple ways of defining new data types, the keyword data is used to declare new data types. The key word deriving is used to derive the data type as an instance of a type class and typically, the standard functions that are supposed to be defined on types of the type class are defined by default. For instance Eq, Ord, Show etc, then you can also define data types with parameters as in the example of Shape, Person, etc.

You have two different types of user defined data types, one you might called the Sum type or union type, where there are multiple constructors on the right. The example here is Day or Shape or the other is the Product type or the struct type that you might be familiar from other languages, where you have only one constructor on the right, the example of this is a Person.