Haskell Tutorial: Datatypes

November 1, 2019

[1]:
```
:opt no-lint
```

1 Datatypes

Sometimes we may want to introduce new data types for specific purposes.

1.1 Employee

We introduce the mechanisms that can be used to define our own datatypes by defining a datatype representing information about an employee.

[2]:
```
data EmployeeInfo = Employee String Int Double [String]
    deriving (Show)

p = Employee "Joe Sample" 22 186.3 []
print p
```

```
Employee "Joe Sample" 22 186.3 []
```

To define a new datatype, use the keyword `data`, which is followed by the name of our new type (also called a type constructor), in our example `EmployeeInfo`. The type constructor is followed by the data constructor `Employee`. The data constructor is used to create a value of the `EmployeeInfo` type. Both the type constructor and the data constructor must start with a capital letter. The `Int`, `Double`, `String` and `[String]` (that follow after the data constructor `Employee`) are the components/fields of the type. (If you are familiar with an object-oriented language, these components serve the same purpose as fields for a class).

In this particular example, the name of the type constructor (`EmployeeInfo`) and the value/data constructor (`Employee`) use a different name for you to see which is which. It is common practice to give them both the same name. There is no ambiguity because the type constructor's name is used only in type declaration / type signatures and the value constructor's name is used in the actual code (in writing expressions).

The `deriving (Show)` part asks Haskell to make sure that the new type belongs to the `Show` typeclass, which implies to derive a `show` function that renders an instance of our newly defined datatype into a string. This `show` function is used by the `print` function. If there would be no `deriving (Show)`, Haskell would not be able to print an instance of our new datatype. We will talk later more about typeclasses later, so lets not dig deeper here.

Consider the fields from the example above. It is not very clear what the `Int`, `Double`, `String` and `[String]` fields mean. To make things as clear as possible, it is possible to introduce synonyms for existing types at any time. Such synonyms give the type a more descriptive name.

[3]:
```
type Age = Int
type Salary = Double
type Name = String
type Manager = [Name]
data EmployeeInfo = Employee Name Age Salary Manager
    deriving (Show)

p = Employee "Joe Sample" 22 86123.35 ["Lucy Boss"]
print p
```

```
Employee "Joe Sample" 22 86123.35 ["Lucy Boss"]
```

Data type definitions can be nested. We improve our example by replacing the `Age` field with a `Birthday` field. Using a `Birthday` in our `Employee` type has the advantage that the birthday is immutable while the age changes once every year. Since a date consists of a year, a month, and a day, we construct another data type to represent dates. (In real projects, you likely want to use a pre-defined date type.)

[4]:
```
type Day = Int
type Month = Int
type Year = Int
data Date = Date Year Month Day
    deriving (Show)

type Name = String
type Birthday = Date
type Salary = Double
type Manager = [Name]

data Employee = Employee Name Birthday Salary Manager
    deriving (Show)

p = Employee "Joe Sample" (Date 1983 06 17) 86123.35 ["Lucy Boss"]
print p
```

```
Employee "Joe Sample" (Date 1983 6 17) 86123.35 ["Lucy Boss"]
```

New data types can also be alternatives of types. The simplest examples are enumerations. We can use alternative types to define the sex of a person.

[5]:
```
data Sex = Female | Male | Diverse
    deriving (Show)
```

The `Sex` datatype has three data constructors, `Female`, `Male` and `Diverse`, separated by a bar symbol (the bar symbol can be read as "or"). The data constructors are commonly referred to as alternatives or cases. The data constructors can take zero or more arguments.

[6]:
```
data Employee = Worker  Name Birthday Sex Salary Manager
              | Manager Name Birthday Sex Salary
    deriving (Show)

w = Worker "Joe Sample" (Date 1983 06 17) Diverse 86123.35 ["Lucy Boss"]
print p
m = Manager "Lucy Boss" (Date 1967 02 20) Female 113213.23
print m
```

```
Employee "Joe Sample" (Date 1983 6 17) 86123.35 ["Lucy Boss"]


Manager "Lucy Boss" (Date 1967 2 20) Female 113213.23
```

Data type definitions can be recursive, giving us the option to have inductively defined data types. Lets make the manager of an empoyee another employee so that we can represent a whole employee hierarchy.

[7]:
```
data Employee = Worker Name Birthday Sex Salary Employee
              | Manager Name Birthday Sex Salary
    deriving (Show)

lucy :: Employee
lucy = Manager "Lucy Boss" (Date 1967 02 20) Female 113213.23

joe :: Employee
joe = Worker "Joe Sample" (Date 1983 06 17) Diverse 86123.35 lucy

print joe
```

```
Worker "Joe Sample" (Date 1983 6 17) Diverse 86123.35 (Manager "Lucy Boss" (Date 1967 2 20) Fema
```

1.2 Maybe

Type definitions can be parameterized by introducing a type parameter in the type definition. A common example is the definitions of `Maybe` in the prelude.

[8]:
```
data Maybe a = Nothing | Just a
    deriving (Show)
```

The definition of `Maybe` has a type as its argument, i.e., the type is parametrized. The `Maybe` type can be used to create types that contain a certain typed value or nothing. This helps in situation where some value is optional or not always defined. Lets take a short excursion by looking at the standard `tail` function: it fails if called with a list that has no tail.

[9]:
```
tail [1..3]
tail [1]
tail []
```

```
[2,3]
```

```
[]
```

```
Prelude.tail: empty list
```

Throwing an error is a pretty bad side effect. With `Maybe`, we have a tool that we can use to write a safe tail function, which returns `Nothing` or `Just` a list.

[10]:
```
safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs

safeTail [1..3]
safeTail [1]
safeTail []
```

```
Just [2,3]
```

```
Just []
```

```
Nothing
```

1.3 Employee (continued)

With this definition of `Maybe`, we can further improve our `Employee` type by enabling managers to be managed by other managers but top-level managers are not managed by anyone. And at this point, we may want to give up the distinction between managers and workers (since an employee is a manager once she manages someone).

[11]:
```
data Employee = Employee Name Birthday Sex Salary (Maybe Employee)
    deriving (Show)

lucy = Employee "Lucy Boss" (Date 1967 02 20) Female 113213.23 Nothing

joe = Employee "Joe Sample" (Date 1983 06 17) Diverse 86123.35 (Just lucy)
```

```haskell
print joe
```

```
Employee "Joe Sample" (Date 1983 6 17) Diverse 86123.35 (Just (Employee "Lucy Boss" (Date 1967 2
```

The interpretation is that the manager is either `Just Employee` or `Nothing`. The `Maybe` type is very widely used to indicate missing values in Haskell.

The last improvement we want to make is to use the record syntax. With the record syntax, we can give the various fields of our type names. Haskell will then automatically generate functions that can be used to access the different fields of a type. The record syntax also changes how data instances are rendered using the `show` function.

[12]:

```haskell
type Day = Int
type Month = Int
type Year = Int
data Date = Date { year :: Year, month :: Month, day :: Day }
    deriving (Show)

type Name = String
type Euro = Double      -- never represent money using floating point types in␣
 ↪real code

data Sex = Female | Male | Diverse
    deriving (Show)

data Employee = Employee { name :: Name
                         , birthday :: Date
                         , sex :: Sex
                         , salary :: Euro
                         , manager :: Maybe Employee }
    deriving (Show)

lucy = Employee { name = "Lucy Boss"
                , birthday = Date { year = 1967, month = 02, day = 20 }
                , sex = Female
                , salary = 113213.23
                , manager = Nothing }

joe = Employee { name = "Joe Sample"
               , birthday = Date { year = 1983, month = 06, day = 17 }
               , sex = Diverse
               , salary = 86123.35
               , manager = Just lucy }

print joe
print (manager joe)
```

```
print lucy
print (manager lucy)
```

```
Employee {name = "Joe Sample", birthday = Date {year = 1983, month = 6, day = 17}, sex = Diverse

Just (Employee {name = "Lucy Boss", birthday = Date {year = 1967, month = 2, day = 20}, sex = Fe

Employee {name = "Lucy Boss", birthday = Date {year = 1967, month = 2, day = 20}, sex = Female,

Nothing
```

1.4 Binary Tree

As another example, let us define a parametric type for a binary tree in order to practice what we have learned.

[13]:
```
data Tree a = Empty
            | Leaf a
            | Branch a (Tree a) (Tree a)
            deriving (Eq, Show)

t = Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)
print t

:type t
```

```
Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)

Tree Char
```

Haskell has derived that `t` is of type `Tree Char`. In the definition of `Tree`, we ask Haskell to derive the typeclasses `Eq` and `Show`. We do this to obtain a `show` function that takes a `Tree` value and converts it into a string so that we can print values. Lets see what happens if we create a Tree of numbers.

[14]:
```
t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
print t

:type t
```

```
Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)

forall a. Num a => Tree a
```

Haskell derived that `s` is of type `Num a => Tree a`, i.e., it is a tree of a type that is constrained to the `Num a` typeclass.

Lets now define a function `values :: Tree a -> [a]` that takes a tree and returns all values stored in the tree as a list. Note that this function works for any type that can be used with the tree type. We can use this function with a tree of strings or a tree of numbers, i.e., this function is polymorphic.

[15]:
```
values :: Tree a -> [a]
values Empty = []
values (Leaf x) = [x]
values (Branch x l r) = values l ++ [x] ++ values r

t = Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)
print $ values t

t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
print $ values t
```

```
"abcdef"
```

```
[1,2,3,4,5,6]
```

The `values` function can be implemented in different ways since the tree may be traversed in three different ways: * pre-order (NLR): 1. Return the value of the current node (N) 1. Return the values of the left subtree (L) 1. Return the values of the right subtree (R) * in-order (LNR): 1. Return the values of the left subtree (L) 1. Return the value of the current node (N) 1. Return the values of the right subtree (R) * post-order (LRN): 1. Return the values of the left subtree (L) 1. Return the values of the right subtree (R) 1. Return the value of the current node (N)

Perhaps we should have three traversal functions.

[16]:
```
preOrderValues :: Tree a -> [a]
preOrderValues Empty = []
preOrderValues (Leaf x) = [x]
preOrderValues (Branch x l r) = [x] ++ preOrderValues l ++ preOrderValues r

inOrderValues :: Tree a -> [a]
inOrderValues Empty = []
inOrderValues (Leaf x) = [x]
inOrderValues (Branch x l r) = inOrderValues l ++ [x] ++ inOrderValues r

postOrderValues :: Tree a -> [a]
postOrderValues Empty = []
postOrderValues (Leaf x) = [x]
postOrderValues (Branch x l r) = postOrderValues l ++ postOrderValues r ++ [x]
```

```haskell
t = Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)
print $ preOrderValues t
print $ inOrderValues t
print $ postOrderValues t
```

```
"dbacfe"
```

```
"abcdef"
```

```
"acbefd"
```

We can also define a function `map :: (a -> b) -> Tree a -> Tree b` applying a function to all values stored in leaves of our tree.

[17]:
```haskell
map :: (a -> b) -> Tree a -> Tree b
map f Empty = Empty
map f (Leaf x) = Leaf (f x)
map f (Branch x l r) = Branch (f x) (map f l) (map f r)

t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
print $ map (+1) t
```

```
Branch 5 (Branch 3 (Leaf 2) (Leaf 4)) (Branch 7 (Leaf 6) Empty)
```

In a similar way, we can define a `foldr` function that takes a function to reduce a tree value and an already reduced value, a zero value, and a tree and returns the reduced value.

[18]:
```haskell
foldr :: (a -> b -> b) -> b -> Tree a -> b
foldr _ z Empty = z
foldr f z (Leaf x) = f x z
foldr f z (Branch x l r) = foldr f (f x (foldr f z r)) l

t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
print $ foldr (+) 0 t
print $ foldr (*) 1 t
```

```
21
```

```
720
```

Note that our in-order `values` function is a special case of a fold over the tree.

[19]:
```haskell
values :: Tree a -> [a]
values = foldr (:) []

t = Branch 'd' (Branch 'b' (Leaf 'a') (Leaf 'c')) (Branch 'f' (Leaf 'e') Empty)
print $ values t
```

```
"abcdef"
```

Since `Tree` is also an instance of the `Eq` typeclass, we can test trees for equality.

[20]:
```haskell
t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
r = map (+1) t
print $ t == r
print $ t /= r
```

```
False
```

```
True
```

[30]:
```haskell
draw :: Show a => Tree a -> String
draw t = pfxDraw "" t where
    pfxDraw p Empty          = p ++ "+:\n"
    pfxDraw p (Leaf x)       = p ++ "+-" ++ show x ++ "\n"
    pfxDraw p (Branch x l r) = center ++ left ++ right where
        center = p ++ "+-" ++ show x ++ "\n"
        left   = pfxDraw (p ++ "  ") l
        right  = pfxDraw (p ++ "  ") r

t = Branch 4 (Branch 2 (Leaf 1) (Leaf 3)) (Branch 6 (Leaf 5) Empty)
putStr $ draw t
```

```
+-4
  +-2
    +-1
    +-3
  +-6
    +-5
    +:
```