So much reading, so little time.

LYAH Chapter 7: Making Our Own Types and Type Classes (40 pages… so long)

Summary:

This chapter is mainly about how you go about creating your own custom types. Types are little labels that values carry so that we can reason about the values (pg.150). You can create your type using the data keyword (e.g. data Bool = False|True). You can write data types with the record syntax instead of naming the field types one after another and separating with spaces, you would use curly brackets. It creates functions that look up fields in the data type. By using deriving keyword, you can make a type an instance of following type classes: Eq, Ord and etc . . .

Notes: (For once I might copy and paste a bit from online version)

Defining a New Data type:

Using **data**, you can define a new data type similar to Bool, Int, Char and yada.
(e.g. data Bool = False | True)
The part before the = denotes the type and after that are the value constructors (specify diff. values that this type can have)
example with Int:
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... | 2147483647

Shaping Up:

Let's say that a shape can be a circle or a rectangle. Here's one possible definition:
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
Value constructors are actually functions that ultimately return a value of a data type. Let's take a look at the type signatures for these two value
constructors.

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Now using the above thingies, here's a function that takes a Shape and returns the area:

```
area :: Shape -> Float
area (Circle _ _ r) = pi * r ^ 2
area (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

The results:

```
ghci> area $ Circle 10 20 10
314.15927
ghci> area $ Rectangle 0 0 100 100
10000.0
```

However, an error would occur if you try to print out Circle 10 20 5, since Haskell doesn't know how to display data type as a string. The use of deriving (Show) at end of data declaration makes that type part of Show type class.
(e.g. data Shape = Circle Float Float Float | Rectangle Float Float Float Float deriving (Show))
This won't actually give you the answer

Improving Shape with the Point Data Type:
Let's make an intermediate
data type that defines a point in two-dimensional space. Then we can use that to make our shapes more understandable.

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

So Circle now has two fields: one type Point and other type Float

```
area :: Shape -> Float
area (Circle _ r) = pi * r ^ 2
area (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

In the Rectangle pattern, we just used nested pattern matching to get the fields of the points.

```
ghci> area (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> area (Circle (Point 0 0) 24)
1809.5574
```

nudge allows you to return a new shape that has the same dimension but located somewhere else

You can also export the shape functions and types in a module by module Shapes(…)

Record Syntax
The book will now try to describe a data type that describes a person

```
data Person = Person String String Int Float String String deriving (Show)
```

Here's one way of grabbing all the information needed for the function

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname
lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname
age :: Person -> Int
age (Person _ _ age _ _ _) = age
height :: Person -> Float
height (Person _ _ _ height _ _) = height
phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number
flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

Here's a better way

```
data Person = Person { firstName :: String
, lastName :: String
, age :: Int
, height :: Float
, phoneNumber :: String
, flavor :: String } deriving (Show)
```

proof that it works!

```
ghci> :t flavor
```

flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String

We don't need to put the fields in the proper order, as long as we list all of them. But if we don't use record syntax, we must specify them in order.

Type Parameters:
Type constructors can take types as parameters to produce new types.
Here's an example of a type they provided
data Maybe a = Nothing | Just a
The a here is the type parameter. And because there's a type parameter involved, we call Maybe a *type constructor*.
Depending on what we want this data type to hold when it's not Nothing, this type constructor can end up producing a type of Maybe Int, Maybe Car, Maybe String, and so on.

If we want to explicitly pass a type as a type parameter, we must do it in the type part of Haskell, which is usually after the :: symbol.
Type parameters are useful because they allow us to make data types that can hold different things.

type of Nothing is Maybe a. Its type is *polymorphic*, which means that it features type variables, namely the a in Maybe a. If some function requires a Maybe Int as a parameter, we can give it a Nothing, because a Nothing doesn't contain a value anyway, so it doesn't matter. The Maybe a type can act like a Maybe Int if it must, just as 5 can act like an Int or a Double.

<u>Should we Parameterize Our Car?</u>
Consider our Car data type:

```
data Car = Car { company :: String
, model :: String
, year :: Int
} deriving (Show)
```

We could change it to this:

```
data Car a b c = Car { company :: a
, model :: b
, year :: c
} deriving (Show)
```

For
instance, given our first definition of Car, we could make a function that displays the car's properties in an easy-to-read format.

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) =
"This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

We could test it like this:

```
ghci> let stang = Car {company="Ford", model="Mustang", year=1967}
```

```
ghci> tellCar stang
"This Ford Mustang was made in 1967"
```

We usually use type parameters when the type that's contained inside the data type's various value constructors isn't really that important for the type to work.
However, it's a very strong convention in Haskell to never add type class constraints in data declarations. Why? Well, because it doesn't provide much benefit, and we end up writing more class constraints, even when we don't need them.

Vector von Doom:
some example implementing vectors

Derived Instances:
Haskell can automatically make our type an instance of any of the following type classes: Eq, Ord, Enum, Bounded, Show, and Read. Haskell can derive the behavior of our types in these contexts if we use the deriving keyword when making our data type.

Equating People:

Consider this data type:

```
data Person = Person { firstName :: String
, lastName :: String
, age :: Int
}
```

It describes a person. Let's assume that no two people have the same combination of first name, last name, and age. If we have records for two people, does it make sense to see if they represent the same person? Sure it does. We can try to equate them to see if they are equal. That's why it would make sense for this type to be part of the Eq type class. We'll derive the instance.

```
data Person = Person { firstName :: String
, lastName :: String
, age :: Int
} deriving (Eq)
```

Of course, since Person is now in Eq, we can use it as the a for all functions that have a class constraint of Eq a in their type signature, such as elem.

Show Me How to Read:
The Show and Read type classes are for things that can be converted to or from strings, respectively. As with Eq, if a type's constructors have fields, their type must be a part of Show or Read if we want to make our type an instance of them.
Let's make our Person data type a part of Show and Read as well.

```
data Person = Person { firstName :: String
, lastName :: String
, age :: Int
} deriving (Eq, Show, Read)
```

Read is pretty much the inverse type class of Show. It's for converting strings to values of our type

Order in the Court!:

We can derive instances for the Ord type class, which is for types that have values that can be ordered. For the purpose of seeing how it behaves when compared, we can think of it as being implemented like this:
data Bool = False | True deriving (Ord)
Because the False value constructor is specified first and the True value constructor is specified after it, we can consider True as greater than False.

In the Maybe a data type, the Nothing value constructor is specified before the Just value constructor, so the value of Nothing is always smaller than the value of Just something, even if that something is minus one billion trillion.

Any Day of the Week:
We can easily use algebraic data types to make enumerations, and the Enum and Bounded type classes help us with that. Consider the following data type:
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday

Because all the type's value constructors are nullary (that is, they don't have any fields), we can make it part of the Enum type class. The Enum type class is for things that have predecessors and successors. We can also make it part of the Bounded type class, which is for things that have a lowest possible value and highest possible value. And while we're at it, let's also make it an instance of all the other derivable type classes.
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
deriving (Eq, Ord, Show, Read, Bounded, Enum)

Because it's part of the Eq and Ord type classes, we can compare or equate days. It's also part of Bounded, so we can get the lowest and highest day. As it's an instance of Enum, we can get predecessors and successors of days and make list ranges from them!

Type Synonyms:
when writing types, the [Char] and String types are equivalent and interchangeable. That's implemented with *type synonyms*. they're just about giving some types different names so that they make more sense to someone reading our code and documentation.
Example: type String = [Char]

Making Our Phonebook Prettier
```
phoneBook :: [(String, String)]
phoneBook =
[("betty", "555-2938")
,("bonnie", "452-2928")
,("patsy", "493-2928")
,("lucille", "205-2928")
,("wendy", "939-8282")
,("penny", "853-2492")
]
```

But you can do it this way instead:
type PhoneBook = [(String,String)]

type PhoneNumber = String
type Name = String
type PhoneBook = [(Name, PhoneNumber)]

So now, when we implement a function that takes a name and a number and checks if that name and number combination is in our phonebook, we can give it a very pretty and descriptive type declaration.
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name, pnumber) `elem` pbook

If we decided not to use type synonyms, our function would have this type:
inPhoneBook :: String -> String -> [(String, String)] -> Bool

Parameterizing Type Synonyms
Type synonyms can also be parameterized. If we want a type that represents an association list type, but still want it to be general so it can use any type as the keys and values, we can do this:
type AssocList k v = [(k, v)]
Now a function that gets the value by a key in an association list can have a type of
(Eq k) => k -> AssocList k v -> Maybe v. AssocList is a type constructor that takes two types and produces a concrete type—for instance, AssocList Int String.

Go Left, Then Right
Another cool data type that takes two types as its parameters is the Either a b type.

Recursive Data Structures:
You can make recursive data types.
Using algebraic data types to implement own list:

data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
This follows our definition of lists. It's either an empty list or a combination of a head with some value and a list. If you're confused about this, you might find it easier to understand in record syntax.
data List a = Empty | Cons { listHead :: a, listTail :: List a} deriving (Show, Read, Eq, Ord)

Improving Our List:
We can define functions to be automatically infix by naming them using only special characters. We can also do the same with constructors, since they're just functions that return a data type. There is one restriction however: Infix constructors must begin with a colon.

infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)

First, notice a new syntactic construct: the fixity declaration, which is the line above our data declaration. When we define functions as operators, we can use that to give them a *fixity* (but we don't have to). A fixity states how tightly the operator binds and whether it's left-associative or right-associative.

Let's Plant a Tree:
They show you how to implement a Binary Search Tree (pg. 133)

Type Classes 102:
Type classes are sort of like interfaces. A type class defines some behavior (such as comparing for equality, comparing for ordering, and enumeration). Types that can behave in that way are made instances of that type class. The behavior of type classes is achieved by defining functions or just type declarations that we then implement. So when we say that a type is an instance of a type class, we mean that we can use the functions that the type class defines with that type.

A Traffic Light Data Type:
data TrafficLight = Red | Yellow | Green
It defines the states of a traffic light. Notice how we didn't derive any class instances for it. That's because we're going to write some instances by hand. Here's how we make it an instance of Eq:

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

We did it by using the instance keyword. So class is for defining new type classes, and instance is for making our types instances of type classes. Because == was defined in terms of /= and vice versa in the class declaration, we needed to overwrite only one of them in the instance declaration. That's called the *minimal complete definition* for the type class—the minimum of functions that we must implement so that our type can behave as the class advertises.
To fulfill the minimal complete definition for Eq, we need to overwrite either == or /=. If Eq were defined simply like this:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Let's make this an instance of Show by hand, too. To satisfy the minimal complete definition for Show, we just need to implement its show function, which takes a value and turns it into a string:

```
instance Show TrafficLight where
show Red = "Red light"
show Yellow = "Yellow light"
show Green = "Green light"
```

Parameterized Types As instances of Type Classes:
Maybe in itself isn't a concrete type—it's a type constructor that takes one type parameter (like Char) to produce a concrete type (like Maybe Char).

When making instances, if you see that a type is used as a concrete type in the type declarations (like the a in a -> a -> Bool), you need to supply type parameters and add parentheses so that you end up with a concrete type.

```
(==) :: (Eq m) => Maybe m -> Maybe m -> Bool
```

This is just something to think about, because == will always have a type of (==) :: (Eq a) => a -> a -> Bool, no matter what instances we make.

A Yes-No Type Class:
let's try to implement this JavaScript-like behavior, just for fun! We'll start out with a class declaration:

```
class YesNo a where
    yesno :: a -> Bool
```

Next up, let's define some instances. For numbers, we'll assume that (as in JavaScript) any number that isn't 0 is true in a Boolean context and 0 is false.

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True
```

Bool itself also holds trueness and falseness, and it's pretty obvious whichis which:

```
instance YesNo Bool where
    yesno = id
```

id is just a standard library function that takes a parameter and returns the same thing.

The Functor Type Class:
Functor type class is for things that can be mapped over. The list type is part of this class.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

fmap takes a function from one type to another and a functor value applied with one type and returns a functor value applied with another type. map is similar:
map :: (a -> b) -> [a] -> [b]
Then examples of Maybe, Trees, Either as a functor.

Kinds and Some Type-Foo:
Type constructors take other types as parameters to eventually produce concrete types. This behavior is similar to that of functions, which take values as parameters to produce values. Also like functions, type constructors can be partially applied. For example, Either String is a type constructor that takes one type and produces a concrete type, like Either String Int.

Types are little labels that values carry so that we can reason about the values. But types have their own little labels called *kinds*. A kind is more or less the type of a type. example:

```
ghci> :k Int
Int :: *
```

What does that * mean? It indicates that the type is a concrete type. A concrete type is a type that doesn't take any type parameters. Values can have only types that are concrete types.

<u>Learn Prolog Now! Ch 1.1: Some Simple Examples</u>

Summary:
Prolog contains three basic constructs: facts, rules and queries, where a collection of facts and rules is called a knowledge base (database). Queries are then used to ask questions about the information held in these knowledge bases. Rules (denoted after the :-) state information that is conditionally true of the situation of interest and variables must be a word that begins with an upper-case letter.

Notes:

- There are only three basic constructs in Prolog: facts, rules, and queries.
- A collection of facts and rules is called a knowledge base (or a database) and Prolog programming is all about writing knowledge bases.
- Use a prolog program by posing queries (questions about the information stored)
- Prolog is useful in computational linguistics and Artificial Intelligence (AI).

Knowledge base 1:

- Knowledge Base 1 (KB1) is simply a collection of facts.
    - Facts are used to state things that are unconditionally true of some situation of interest.

For example, we can state that Mia, Jody, and Yolanda are women, that Jody plays air guitar, and that a party is taking place, using the following five facts:

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
```

Here are some examples. We can ask Prolog whether Mia is a woman by posing the query:

```
?- woman(mia).
```

Prolog will answer

```
yes
```

the ?- symbol is the prompt symbol that the Prolog interpreter displays when it is waiting to evaluate a query.

Knowledge Base 2:

Here is KB2, our second knowledge base:

```
happy(yolanda).
listens2Music(mia).
listens2Music(yolanda):- happy(yolanda).
playsAirGuitar(mia):- listens2Music(mia).
playsAirGuitar(yolanda):- listens2Music(yolanda).
```

- The last three items it contains are rules.
    - Rules state information that is conditionally true of the situation of interest.
    - the first rule says that Yolanda listens to music if she is happy
    - the :- should be read as “if”, or “is implied by”.
        - The part on the left hand side of the :- is called the head of the rule
        - the part on the right hand side is called the body

- If a knowledge base contains a rule head :- body, and Prolog knows that body follows from the information in the knowledge base, then Prolog can infer head.
    - This fundamental deduction step is called modus ponens.

Let's consider an example. Suppose we ask whether Mia plays air guitar:

```
?- playsAirGuitar(mia).
```

Prolog will respond yes. Why? Well, although it can't findplaysAirGuitar(mia) as a fact explicitly recorded in KB2, it can find the rule

```
playsAirGuitar(mia):- listens2Music(mia).
```

Moreover, KB2 also contains the fact listens2Music(mia) . Hence Prolog can use the rule of modus ponens to deduce thatplaysAirGuitar(mia) .

- The facts and rules contained in a knowledge base are called clauses.
    - Another way of looking at KB2 is to say that it consists of three predicates (or procedures)
        - listens2Music, happy, and playsAirGuitar
    - We can view a fact as a rule with an empty body.

Knowledge Base 3:

KB3, our third knowledge base, consists of five clauses:

```
happy(vincent).
listens2Music(butch).
playsAirGuitar(vincent):-
    listens2Music(vincent),
    happy(vincent).
playsAirGuitar(butch):-
    happy(butch).
playsAirGuitar(butch):-
    listens2Music(butch).
```

- two items in body, or two goals
    - the comma , that separates the goal listens2Music(vincent)and the goal happy(vincent) in the rule's body.
    - the way logical conjunction is expressed in Prolog
    - Both condition needs to be cleared for Prolog to say yes

Knowledge base 4:

Here is KB4, our fourth knowledge base:

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent,mia).
loves(marsellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).
```

the novelty this time lies not in the knowledge base, it lies in the queries we are going to pose. In particular, for the first time we're going to make use of variables . Here's an example:
?- woman(X).

- That is, this query asks Prolog: tell me which of the individuals you know about is a woman.
- The X is a variable
  - any word beginning with an upper-case letter is a Prolog variable
  - it's a placeholder for information
- Remember that ; means or , so this query means: are there any alternatives ?

Prolog returns the answer

```
X = mia ;
X = jody ;
X = yolanda
```

Let's try a more complicated query, namely

```
?- loves(marsellus,X), woman(X).
```

Remember that , means and , so this query says: is there any individual X such that Marsellus loves X and X is a woman ?
The answer is Mia.

Knowledge Base 5:

the knowledge base KB5:

```
loves(vincent,mia).
loves(marsellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).

jealous(X,Y):- loves(X,Z), loves(Y,Z).
```

KB5 contains four facts about the loves relation and one rule.
it is defining a concept of jealousy. It says that an individual X will be jealous of an individual Y if there is some individual Z that X loves, and Y loves that same individual Z too.

Suppose we pose the query:

```
?- jealous(marsellus,W).
```

This query asks: can you find an individual W such that Marsellus is jealous of W ? Vincent is such an individual.