User-Define Types and Type Classes 1

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

User-Defined Types

- In the standard library, the `Bool` type is defined as follows:

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

- Use the `data` keyword to define a new typepause
- `Bool` is the type name and it must start with uppercase letter
- Think of `|` as “or”
- `False` and `True` are value constructors and they must start with uppercase letter

Shape

- Suppose we want to create a new data type named `Shape`
  - It consists of either `Circle` or `Rectangle`
  - Properties of a circle are coordinate of its center and its radius
  - Properties of a rectangle are coordinates of its top-left and bottom-right corner
- One way to define this new data type is as follows:

```
-- Shape.hs
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

- Once loaded into `GHCi`, we can view some information:

```
ghci> :info Shape
data Shape
  = Circle Float Float Float | Rectangle Float Float Float Float
        -- Defined at Shape.hs:3:1
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Functions on `Shape`

- Note that types of `Circle` and `Rectangle` look like functions
- They are value constructors
- They are used to construct values of type `Shape`

```
ghci> :t Circle 1.2 3.4 5.6
Circle 1.2 3.4 5.6 :: Shape
ghci> :t Rectangle 1 2 3 4
Rectangle 1 2 3 4 :: Shape
```

- Let's define the new function `area` for shapes:

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

- Note that we can pattern match against value constructors
- Some tests:

```
ghci> area (Circle 0 0 1)
3.1415927
ghci> area (Rectangle 1 2 3 4)
4.0
```

Deriving Show

- To be able to **show** a value of this type, it must be an instance of the `Show` type class
- For simplicity, we can use **automatic derive**:

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

- Now we can do the following:

```
ghci> Circle 0 0 1.2
Circle 0.0 0.0 1.2
ghci> Rectangle 1 2 3 4
Rectangle 1.0 2.0 3.0 4.0
ghci> "The shape is " ++ show (Circle 0 0 1.2)
"The shape is Circle 0.0 0.0 1.2"
```

Point

- The center of a circle or corners of a rectangle are points (2D coordinates)
- Let's create a new data type for points and use it in `Shape`

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

- We also need to modify our `area` function:

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

Translate

- Suppose we want to translate our shapes to a new location using the amounts to move on the x-axis and y-axis
- Let's define the `translate` function:

```
translate :: Shape -> Float -> Float -> Shape
translate (Circle (Point x y) r) a b =
     Circle (Point (x + a) (y + b)) r
translate (Rectangle (Point x1 y1) (Point x2 y2)) a b =
     Rectangle (Point (x1 + a) (y1 + b)) (Point (x2 + a) (y2 + b))
```

- Some tests:

```
ghci> translate (Circle (Point 1 2) 1.5) 2 (-3)
Circle (Point 3.0 (-1.0)) 1.5
ghci> translate (Rectangle (Point 1 2) (Point 3 4)) (-3) 2
Rectangle (Point (-2.0) 4.0) (Point 0.0 6.0)
```

Simple Shapes

- A simple circle is a circle with radius `r` and its center is at the origin
- A simple rectangle is a rectangle with a width `w` and height `h` and its top-left corner is at the origin
- Let's create functions for creating simple circles and rectangles:

```
simpleCircle r = Circle (Point 0 0) r
simpleRectangle w h = Rectangle (Point 0 0) (Point w h)
```

- Let's try:

```
ghci> simpleCircle 4.3
Circle (Point 0.0 0.0) 4.3
ghci> simpleRectangle 2 3
Rectangle (Point 0.0 0.0) (Point 2.0 3.0)
```

Export A Module with Types

- We can export our newly created module by inserting the following lines at the top of our `Shape.hs`

```
-- Shape.hs
module Shape
( Point (..),
  Shape (..),
  area,
  translate,
  simpleCircle,
  simpleRectangle)
where
  :
```

- We use `(..)` to export all value constructors of `Point` and `Shape` types

Export A Module with Types

- Sometimes, we may not want to export our value constructors
- This can be done by removing `(..)` after a type name

```
-- Shape.hs
module Shape
( Point (..),
  Shape,
  area,
  :
```

- In doing so, a user cannot construct a value of type `Shape` directly
- They have to use either `simpleCircle` or `simpleRectangle` functions
- A user will not be able to pattern match against constructors of `Shape`
- This makes the `Shape` data type more abstract since its implementation is hidden

Record Syntax

- Suppose we want to create a value about a person that contains the following information:
  - First name of type `String`
  - Last name of type `String`
  - Height of type `Float`
  - Address of type `String`
  - Date of birth of type `String`
  - Phone numbers of type `[String]`
- One way is to create a new data type named `Person` as follows:

```
data Person = Person String String Float String String [String]
              deriving (Show)
```

- Now, we can create a value of type `Person` as follows:

```
ghci> Person "Kyle" "Root" 5.2 "Madison, WI 53703" "12/03/1940"
                         ["6083083662"]
Person "Kyle" "Root" 5.2 "Madison, WI 53703" "12/03/1940"
["6083083662"]
```

- Not too bad but almost unreadable if you do not know the structure of the `Person` data type

Record Syntax

- Now, if we want functions to extract each piece of information of a person:

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

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

address :: Person -> String
address (Person _ _ _ addr _ _) = addr

dateOfBirth :: Person -> String
dateOfBirth (Person _ _ _ _ dob _) = dob

phoneNumbers :: Person -> [String]
phoneNumbers (Person _ _ _ _ _ ns) = ns
```

Record Syntax

- With functions defined earlier, we can perform the following:

```
ghci> aGuy = Person "Kyle" "Root" 5.2 "Madison, WI 53703"
                    "12/03/1940" ["6083083662"]
ghci> address aGuy
"Madison, WI 53703"
ghci> firstName aGuy
"Kyle"
ghci> height aGuy
5.2
```

- This is fine but there is a better way using the **record syntax**

```
data Person = Person { firstName :: String,
                       lastName :: String,
                       height :: Float,
                       address :: String,
                       dateOfBirth :: String,
                       phoneNumbers :: [String]}
            deriving (Show)
```

Record Syntax

- In doing so, field names become functions

```
ghci> aGuy = Person { firstName = "Kyle", lastName = "Root",
                      height = 5.2, address = "Madison, WI 53703",
                      dateOfBirth = "12/03/1940",
                      phoneNumbers = ["6083083662"]}
ghci> :t address
address :: Person -> String
ghci> address aGuy
"Madison, WI 53703"
ghci> firstName aGuy
"Kyle"
ghci> height aGuy
5.2
```

- When we drive `Show` the display contains more information:

```
ghci> aGuy
Person {firstName = "Kyle", lastName = "Root", height = 5.2,
        address = "Madison, WI 53703", dateOfBirth = "12/03/1940",
        phoneNumbers = ["6083083662"]}
```

Type Parameter

- A value constructor takes a number of arguments and return a new value of a specific type
- Examples:
  - `Point` $\rightsquigarrow$ `Point`
  - `Circle` and `Rectangle` $\rightsquigarrow$ `Shape`
  - `Person` $\rightsquigarrow$ `Person`
- **Type constructors** take types as arguments and return new types
- Recall the data type `Maybe`

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

- `a` is a type variable
- `Maybe` is the type constructor
- `Nothing` and `Just` are value constructors
- `Maybe Int` and `Maybe [Char]` are types

Type Parameter

- We usually do not provide the type parameter

```
ghci> :t Just 'a'
Just 'a' :: Maybe Char
```

- Because of the type inference, Haskell know that `Just 'a'` has type `Maybe Char`
- However, `Just 5` does not have enough information:

```
ghci> :t Just 5
Just 5 :: Num a => Maybe a
```

- Since `5` can be one of an instance of the `Num` type class
- We can be a bit more specific:

```
ghci> :t Just 5 :: Maybe Int
Just 5 :: Maybe Int :: Maybe Int
```

- List is another type with type parameter
- Type parameters are useful because they allow us to make data types that can hold different things

Create Our Own List

- Let's try to create our own list data type with type parameter

```
data MyList a = EmptyList
              | Cons a (MyList a)
              deriving (Show)
```

  - `MyList` is a type constructor
  - `EmptyList` and `Cons` are value constructors

- Here are some values of type `MyList a`

```
ghci> Cons 'a' (Cons 'b' (Cons 'c' EmptyList))
Cons 'a' (Cons 'b' (Cons 'c' EmptyList))
ghci> Cons 5 (Cons 12 EmptyList)
Cons 5 (Cons 12 EmptyList)
```

- Let's check some types:

```
ghci> :t Cons 'a' EmptyList
Cons 'a' EmptyList :: MyList Char
ghci> :t Cons "Hello" EmptyList
Cons "Hello" EmptyList :: MyList [Char]
```

- With the parameter, we can create our own list of any types

Deriving the `Eq` Type Class

- Haskell can automatically make our type an instance of any of the following type classes: `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read`
- Here is an example of deriving the `Show` and `Eq` type classes

```
data Person = Person String String Int
            deriving (Show, Eq)
```

- Deriving the `Eq` type class allows operators == and /= to be used with values of our `Person` type
- With deriving the `Eq` type class, we can compare two persons:

```
ghci> (Person "John" "Smith" 52) == (Person "John" "Doe" 52)
False
ghci> (Person "John" "Smith" 52) == (Person "John" "Smith" 52)
True
```

- Since we use `String` and `Int` in our `Person` type, they **must** be instances of the `Show` and `Eq` type classes and they are

Deriving the `Read` Type Class

- The `read` function allows us to read a string into a type
- Let's derive the `Read` type class:

```
data Person = Person String String Int
            deriving (Show, Eq, Read)
```

- Now we can read a string into our `Person` type:

```
ghci> aString = "Person \"John\" \"Smith\" 52"
ghci> read aString :: Person
Person "John" "Smith" 52
```

- We need to supply the type in the above expression since `read` does not know what type it should be
- But this one is okay:

```
ghci> read aString /= (Person "John" "Doe" 52)
True
```

- We do not need to give it a type in the above example because of type inference

Deriving the Ord Type Class

- If we want to compare values of our type using comparison operator or the `compare` function, we need to derive the `Ord` type class
- Here is an example:

```
data Date = Sun | Mon | Tue | Wed | Thu | Fri | Sat
          deriving (Show, Eq, Ord)
```

- Now we can compare values in our type:

```
ghci> Mon < Thu
True
ghci> Sun > Sat
False
ghci> compare Mon Tue
LT
```

- For automatic deriving, the order is based on the order of declarations of your value constructors

Deriving the Bounded and Enum Type Classes

- For an enumerate type like our `Date`, highest and lowest values exists
- By deriving `Bounded` and `Enum`, they allow us to use list range
- Here is an example:

```
data Date = Sun | Mon | Tue | Wed | Thu | Fri | Sat
          deriving (Show, Eq, Ord, Bounded, Enum)
```

- Examples:

```
ghci> minBound :: Date
Sun
ghci> maxBound :: Date
Sat
ghci> [Mon .. Thu]
[Mon,Tue,Wed,Thu]
ghci> [Sat,Fri ..]
[Sat,Fri,Thu,Wed,Tue,Mon,Sun]
ghci> [Tue ..]
[Tue,Wed,Thu,Fri,Sat]
ghci> succ Mon
Tue
```

Type Synonyms

- Consider a simple phone book using a list of pairs of strings:

```
ghci> phoneBook = [("Alice", "351-2934"), ("Bob", "274-9924"),
                   ("Carol", "598-4498"), ("David", "245-5137")]
ghci> :t phoneBook
phoneBook :: [([Char], [Char])]
```

- The above phone has type `[([Char], [Char])]`
- A type synonyms allow us to convey some information:

```
type PhoneBook = [([Char], [Char])]

listAllNumber :: PhoneBook -> [[Char]]
listAllNumber [] = []
listAllNumber (x:xs) = snd x : listAllNumber x
```

- It allows us to use `PhoneBook` instead of `[([Char],[Char])]`
- We can also parameterize a type synonym:

```
type MyLazyPair a b = (a, b)

justFirst :: MyLazyPair a b -> a
justFirst (x, _) = x

justSecond :: MyLazyPair a b -> b
justSecond (_, y) = y
```

The `Either` Type

- Let's look at an interesting type that takes two types as parameters:

```
data Either a b = Left a
                | Right b
                deriving (Eq, Ord, Read, Show)
```

- It can be used to encapsulate a value of one type or another
  - `Left` with a content of type `a`
  - `Right` with a content of type `b`
- Let's compare this to the `Maybe` data type

```
data Maybe a = Nothing
             | Just a
             deriving (Eq, Ord, Read, Show)
```

- We usually use `Nothing` to represent a failed computation
  - Unfortunately, `Nothing` does not tell you anything problem
  - If there is only one chance that fails, it is fine to use `Nothing`

The `Either` Type

- The `Either` data type can help us by:
  - `Left` with some kind of error message
  - `Right` with value for a successful computation
- Suppose we use a `(num, status, code)` to represent a locker where
  - `num` is the locker number
  - `status` is either `"Taken"` or `"Free"`
  - `code` is the combination for the locker `num`
- Here is our simple database:

```
lockerDB = [(101, "Taken", "12-34-56"),
            (102, "Free", "21-43-24"),
            (104, "Free", "11-22-33"),
            (105, "Taken", "34-22-41")]
```

- Note that locker number 103 is missing because it is broken.

The `Either` Type

- The `requestLocker` function should return the code of the given locker number if it is free
- It will not return the code if
  - The locker is currently taken or
  - The locker is broken
- Here is the code:

```
requestLocker num [] = Left "Locker is out-of-service"
requestLocker num ((n,s,c):xs)
    | n == num && s == "Free"   = Right c
    | n == num && s == "Taken" = Left "The locker is taken"
    | otherwise                = requestLocker num x
```

- Here is a couple tests:

```
ghci> requestLocker 102 lockerDB
Right "21-43-24"
ghci> requestLocker 103 lockerDB
Left "Locker is out-of-service"
ghci> requestLocker 105 lockerDB
Left "The locker is taken"
```

Recursive Data Structures

- We already see a couple of recursive data structures:

```
data MyList a = EmptyList
              | Cons a (MyList a)
              deriving (Show)

data BTree a = EmptyBTree
             | BTree a (BTree a) (BTree a)
             deriving (Show)
```

- If you want to use an infix operator as a value constructor
  - The name must start with `:`
  - Declare the precedence value using `infixr` or `infixl`
- For example, instead of `Cons`, we are going to use `:-:`

```
infixr 5 :-:
data MyList a = EmptyList
              | a :-: (MyList a)
              deriving (Show)
```

- Now, we can do something like the following:

```
ghci> :t 'a' :-: 'b' :-: EmptyList
'a' :-: 'b' :-: EmptyList :: MyList Char
```