Recollecting Haskell, Part IV

User Defined Types

CIS 352

Programming Languages

January 22, 2019

Enumerated Types, 1

Recall type synonyms:

```
type Point = (Float,Float) -- A shorthand name for a type
```

You also have many means of creating new types. E.g.,

```
data Season = Winter | Spring | Summer | Fall
```

- This is called an **enumerated type**.
- We can use `Season` just like any other type. E.g.,

```
hasSnow :: Season -> Bool
hasSnow Summer = False
hasSnow _      = True
```

!!! **However**, there are problems with our definition. E.g.,

- Haskell doesn't know how to print values of type Season
- Haskell doesn't know how to compare values of type Season
- Etc.

Enumerated Types, 2

```
data Season = Winter | Spring
            | Summer | Fall
```

!!! Haskell doesn't know how to (print, compare, ...) Season-values.

Quick fix. Change the definition to:

```
data Season = Winter | Spring | Summer | Fall
              deriving (Eq,Ord,Show)
```

Now, the following work just fine:

```
    Winter == Winter
    succ Winter
    Winter < Summer
```

What is the magic?

- $\boxed{\texttt{deriving (Eq,Ord,Show)}}$ joins up the just defined type (Season) to type classes Eq, Ord, Show with default definitions.
- E.g., for Season the derived Ord-ordering is

  `Winter < Spring < Summer < Fall`

Class Exercise: Rock-Paper-Scissors

```
data Move   = Rock | Paper | Scissors
              deriving (Show,Eq)

data Result = Win1 | Win2 | Tie
              deriving (Show,Eq)

game :: Move -> Move -> Result
          ???
```

Product Types

Another sort of DIY (Do It Yourself) type:

```
data Location = Address Int String
                deriving (Show)

nextDoor :: Location -> Location
nextDoor (Address num street) = Address (num+1) street

showAddr :: Location -> String
showAddr (Address num street) = (show num) ++ " " ++ street
```

We could have defined:

```
type LocationToo = (Int,String)
```

Pros of Location

Many things can be of type `(Int,String)`,
but a `Location` *is labeled as an address—so hard to confuse.*

Pros of `LocationToo`

All the tuple stuff (e.g., `fst`, `zip`, *...) works for* `LocationToo`

Aside: Record Types, 1

A street address as a product type

```
data Location = Address Int String
                deriving (Eq,Show)
```

A street address as a record type

```
data Location’ = Address’ { number :: Int ,
                            street :: String }
                 deriving (Eq,Show)
```

What do we gain?

```
ghci> let wh = Address’ 1600 "Penn. Ave."
ghci> wh
Address’ number = 1600, street = "Penn. Ave."
ghci> :t number
number :: Location’ -> Int
ghci> number wh
1600
ghci> street wh
"Penn. Ave."
```

Aside: Record Types, 2

A street address as a record type

```
data Location' = Address' { number :: Int, street :: String }
                 deriving (Eq,Show)
```

```
ghci> let baxter = Address' { street = "East 42nd Street",
              number = 39}

ghci> baxter {number=100}
Address' {number = 100, street = "East 42nd Street"}

ghci> baxter
Address' {number = 39, street = "East 42nd Street"}
```

- So you have getters and "setters" if you need them. **(Why the scare quotes?)**
- Handy for data-types with lots of fields.
- **Do not use these to avoid pattern matching!!!!** **(Why the fuss?)**
- See Chapter 7 of LYAH for more details.

Making a Type an Instance of a Type Class, 1

Consider

```
-- Time h m represents a time Zeit of h hours & m mins
data Zeit = Time Integer Integer
```

Making `Zeit` an instance of `Eq`

```
instance Eq Zeit where
    Time h1 m1 == Time h2 m2  =  (60*h1+m1==60*h2+m2)
```

Now:

- `Time 0 20 == Time 0 20` $\rightsquigarrow$ `True`
- `Time 1 20 == Time 0 80` $\rightsquigarrow$ `True`
- `Time 1 21 /= Time 0 80` $\rightsquigarrow$ `True`

Making `Zeit` an instance of `Ord`

```
instance Ord Zeit where
    Time h1 m1 <= Time h2 m2  =  (60*h1+m1 <= 60*h2+m2)
```

Making a Type an Instance of a Type Class, 2

```
-- Time h m represents a time Zeit of h hours & m mins
data Zeit = Time Integer Integer
```

Making Zeit an instance of Num

```
instance Num Zeit where
    Time h1 m1 + Time h2 m2  =  Time h m
        where (h,m) = quotRem (60*(h1+h2)+m1+m2) 60
    Time h1 m1 - Time h2 m2  =  Time h m
        where (h,m) = quotRem (60*(h1-h2)+m1-m2) 60
    fromInteger n = Time h m
        where (h,m) = quotRem  n 60
```

Making Zeit an instance of Show

```
instance Show Zeit where
    show (Time h m)
        =  show h ++ " hours and " ++ show m ++ " minutes"
```

More later

Class Exercise: Complex Numbers, 1

Complex Numbers (see http://en.wikipedia.org/wiki/Complex_number)

```
data Cmplx = Cmplx Double Double      -- Cmplx a b ≡ a+bi
```

```
re, im :: Cmplx -> Double
    ???
```

```
instance Show Cmplx where
    show (Cmplx x y) = show x ++ "+" ++ show y ++ "i"
```

```
instance Eq Cmplx where
    ???
```

```
instance Ord Cmplx where
    ???
```

Exercise: Complex Numbers, 2

Complex Arithmetic (see http://en.wikipedia.org/wiki/Complex_number)

$$\begin{aligned}(x_1 + y_1 i) + (x_2 + y_2 i) &= (x_1 + x_2) + (y_1 + y_2)i. \\ (x_1 + y_1 i) \cdot (x_2 + y_2 i) &= (x_1 \cdot x_2 - y_1 \cdot y_2) + (x_1 \cdot y_2 + x_2 \cdot y_2)i. \\ &\vdots\end{aligned}$$

```
data Cmplx = Cmplx Double Double      Cmplx a b ≡ a+bi
```

```
instance Num Cmplx where
    ???
```

For the standard Haskell complex-numbers package, see: http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Complex.html

Sum Types

```
type Point = (Float,Float)                          -- not the same as LYAH's
data Shape = Circle Point Float | Rectangle Point Point
             deriving (Show)
-- Circle p r       = a circle with center p and radius r
-- Rectangle p1 p2  = a rectangle with opposite corner pts p1 and p2
```

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

circum (Circle _ r)                = 2 * pi * r
circum (Rectangle (x1,y1) (x2,y2)) = 2 * (abs(x1-x2) + abs(y1-y2))
```

```
-- nudge s (x,y) = shape s moved by the vector (x,y)
nudge :: Shape -> Point -> Shape
nudge (Circle (x,y) r) (x',y')
    = Circle (x+x',y+y') r
nudge (Rectangle (x1,y1) (x2,y2)) (x',y')
    = Rectangle (x1+x',y1+y') (x2+x',y2+y')
```

General Form of Algebraic Types

$$
\begin{array}{lll}
\texttt{data Typename} & = & \texttt{Constr}^A\ \texttt{t}_1^A\ \ldots \texttt{t}_k^A \\
 & \mid & \texttt{Constr}^B\ \texttt{t}_1^B\ \ldots \texttt{t}_\ell^B \\
 & & \vdots
\end{array}
$$

where

- Typename can take parameters *(more on this later)*
- $\texttt{Constr}^A$, $\texttt{Constr}^B$, … are constructor names
- $\texttt{t}_i^A$, $\texttt{t}_j^B$, … are types, and
- the definitions can be recursive.

Example: A DIY list type

```
data IntList = Empty | Cons Int IntList
               deriving (Show,Eq,Ord)
```

DIY Int Lists, Continued

Example: A DIY list type

```
data IntList = Emtpy | Cons Int IntList
               deriving (Show,Eq,Ord)
```

```
-- Convert from IntLists to convential list of Ints
convert :: IntList -> [Int]
convert Empty       = []
convert (Cons x xs) = x:(convert xs)

-- Convert from convential list of Ints to IntLists
revert :: [Int] -> IntList
revert []     = Empty
revert (x:xs) = Cons x (revert xs)
```

What about a general DIY list data type?

Parameterized Data Type Definitions

You can parameterize an algebraic type by type params.

A DIY general list data type

```
data MyList a = Empty’ | Cons’ a (MyList a)
                deriving (Eq, Show)

convert’ :: MyList a -> [a]
convert’ Empty’       = []
convert’ (Cons’ x xs) = x:(convert’ xs)

revert’ :: [a] -> MyList a
revert’ []     = Empty’
revert’ (x:xs) = Cons’ x (revert’ xs)
```

Making Zeit an *Abstract Data Type*, 1

Zeit.hs

```
module Zeit (Zeit(..),stretch) where

data Zeit = Time Integer Integer

-- Convert Zeits to minutes  (not exported)
toMins :: Zeit -> Integer
toMins (Time h m) = 60*h+m

-- Stretch t f = the Zeit t stretched by amount f
-- E.g.: stretch (Time 1 0) 1.5  = Time 1 30
stretch :: Zeit -> Float -> Zeit
stretch t s = fromInteger(round(s * fromIntegral(toMins t)))

instance Eq Zeit where
    t1 == t2  =  toMins t1 == toMins t2
instance Ord Zeit where
    t1 < t2   =  toMins t1 < toMins t2
```

Making Zeit an *Abstract Data Type*, 2

```
-- Zeit.hs  continued

instance Num Zeit where
    t1 + t2       = fromInteger (toMins t1 + toMins t2)
    t1 - t2       = fromInteger (toMins t1 - toMins t2)
    abs t         = fromInteger(abs(toMins t))
    t1 * t2       = error "(*) not defined for Zeit"
    signum t      = error "signum not defined for Zeit"
    fromInteger n = Time h m
        where (h,m) = divMod n 60

instance Show Zeit where
    show (Time h m) = show h ++ " hours and "
                        ++ show m ++ " minutes"
```

Digression on Importing Modules, 1

- importing all of a module

```
import Data.List
```

- importing select items from a module

```
import Data.List (nub,union)
```

- importing *all but* select items from a module

```
import Data.List hiding (nub,sort)
```

Digression on Importing Modules, 2

- a qualified import *(to avoid name clashes)*

```
import qualified Data.Map.Strict        includes a function named null
 ...
    if null lst then ...                                 the standard null
 ...
    if Data.Map.Strict.null table then ...                     Map's null
```

- a qualified import with a shorthand prefix

```
import qualified Data.Map.Strict as M
 ...
    if null lst then ...                                 the standard null
 ...
    if M.null table then ...                                   Map's null
```

See LYAH Chapter 6 for more details and some nice examples.

...now back to user defined types

Back to Algebraic Data Types

The `Maybe` Type ≈ (a way of adding a "bottom" value to a type)

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

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

```
lookup "Penny" [("Gomez",2),("Dixie",7),("Penny",10)]
   ⤳  Just 10

lookup "Dixie" [("Gomez",2),("Dixie",7),("Penny",10)]
   ⤳  Just 7

lookup "Gaspode" [("Gomez",2),("Dixie",7),("Penny",10)]
   ⤳  Nothing
```

The Rust guys really like maybe (Option) types, see:
https://doc.rust-lang.org/book/ch10-01-syntax.html?highlight=option#in-enum-definitions

Adding Maybe to Some Type Classes

The Maybe Type

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

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y     = x == y
    Nothing == Nothing   = True
    _ == _               = False
```

```
ghci> :i Maybe
data Maybe a = Nothing | Just a  -- Defined in Data.Maybe
instance Eq a => Eq (Maybe a) -- Defined in Data.Maybe
instance Monad Maybe -- Defined in Data.Maybe
instance Functor Maybe -- Defined in Data.Maybe
instance Ord a => Ord (Maybe a) -- Defined in Data.Maybe
instance Read a => Read (Maybe a) -- Defined in GHC.Read
instance Show a => Show (Maybe a) -- Defined in GHC.Show
instance Arbitrary a => Arbitrary (Maybe a)
  -- Defined in Test.QuickCheck.Arbitrary
```

Back to Recursive Types

See LYAH's working out of the `List` and `Tree` types.

A Type for Propositional Logic

```
type Name = String
data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :|: Prop
          | Prop :&: Prop
          deriving (Eq, Ord)

type Names = [Name]
type Env = [(Name, Bool)]
```

at this point we switch to emacs

References

- Wikipedia's article on algebraic data types:
  http://en.wikipedia.org/wiki/Algebraic_data_type
- LYAH: Making Our Own Types and Typeclasses:
  http://learnyouahaskell.com/making-our-own-types-and-typeclasses
- Jeremy Gibbons: Calculating Functional Programs:
  http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/acmmpc-calcfp.pdf[1]
  (Explains some of the theory behind algebraic data types.)

[1] From Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of Lecture Notes in Computer Science. Springer-Verlag, 2002.