Recalling Haskell, Part IV
User Defined Types

CIS 352/Spring 2015
Programming Languages
January 20, 2015

Enumerated Types, 1
Recall type synonyms:

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

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

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

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

```haskell
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

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

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

Quick fix. Change the definition to:

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

Now, the following work just fine:

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

What is the magic?
- `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

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

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

game :: Move -> Move -> Result
```
Product Types

Here is another form of DIY type:

```haskell
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:

```haskell
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

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

A street address as a record type

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

What do we gain?

*Main> let wh = Address' 1600 "Penn. Ave."
*Main> wh
Address' number = 1600, street = "Penn. Ave."
*Main> :t number
number :: Location' -> Int
*Main> number wh
1600
*Main> street wh
"Penn. Ave."

Aside: Record Types, 2

A street address as a record type

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

```haskell
*Main> let baxter = Address' { street = "East 42nd Street",
    number = 122}
*Main> baxter {number=300}
Address' {number = 300, street = "East 42nd Street"}
*Main> baxter
Address' {number = 122, 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 LYHFGG for more details.

Making a Type an Instance of a Type Class, 1

Consider

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

Making Zeit an instance of Eq

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

Now:

* Time 0 20 == Time 0 20 ~ True
* Time 1 20 == Time 0 80 ~ True
* Time 1 21 /= Time 0 80 ~ True

Making Zeit an instance of Ord

```haskell
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

Exercise: Complex Numbers, 2

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

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

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

\[
(x_1 + y_1 i) + (x_2 + y_2 i) = (x_1 + x_2) + (y_1 + y_2)i.
\]
\[
(x_1 + y_1 i) \cdot (x_2 + y_2 i) = (x_1 \cdot x_2 - y_1 \cdot y_2) + (x_1 \cdot y_2 + x_2 \cdot y_2)i.
\]

instance Num Cmplx where ???

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

instance Eq Cmplx where ???

instance Ord Cmplx where ???

For the standard Haskell complex-numbers package, see: http://hackage.haskell.org/package/base-4.7.0.2/docs/Data-Complex.html

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 ???

Sum Types

type Point = (Float,Float)

data Shape = Circle Point Float | Rectangle Point Point

deriving (Show)

-- Circle p r = a circle with center p and radius r
deriving (Show)

-- Rectangle p1 p2 = a rectangle with opposite corner pts p1 and p2

data Shape = Circle Point Float | Rectangle Point Point

deriving (Show)

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')
Algebraic Types

General Form of Algebraic Types

```
data Typename = Constr^A t_1^A ... t_k^A
  | Constr^B t_1^B ... t_\ell^B
  |
where

  - Typename can take parameters (more on this later)
  - Constr^A, Constr^B,... are constructor names
  - t_i^A, 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)
```

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

```
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 = fromIntegral(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

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

- a qualified import with a shorthand prefix
  import qualified Data.Map 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
  if null lst then ...
  ...
  if Data.Map.null table then ...
  Map's null

  ~> Just 6

  ~> Just 13

  ~> Nothing

The Rust guys really like maybe types, see:
http://doc.rust-lang.org/book/generics.html and
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

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)]

References

- Wikipedia’s article on algebraic data types: http://en.wikipedia.org/wiki/Algebraic_data_type
  (Explains some of the theory behind algebraic data types.)