CPSC 312

Functional and Logic Programming

November 26, 2015

Revisiting Algebraic Types

```
data Season = Spring | Summer | Autumn | Winter
  deriving (Eq, Ord, Enum, Show, Read)

data People = Person Name Age
or if you will:
data Person = Person Name Age
```

Recursive Types

```
data Expr =            -- An expression can be either
      Lit Integer |     -- a literal integer
      Add Expr Expr |  -- an addition with two expressions
      Sub Expr Expr   -- or a subtraction with two expressions
```

Examples:

```
Lit 0
Add (Lit 5) (Lit 3)
Add (Sub (Lit 7) (Lit 3)) (Sub (Lit 3) (Lit 7))
```

Recursive Types

Recursive type definition makes it easy to define common functions for types, using pattern matching:

```
eval :: Expr -> Integer
eval (Lit n) = n
eval (Add expr1 expr2) = (eval expr1) + (eval expr2)
eval (Sub expr2 expr2) = (eval expr1) - (eval expr2)
```

Recursive Types

Recursive type definition makes it easy to define common functions for types, using pattern matching:

```
eval :: Expr -> Integer
eval (Lit n) = n
eval (Add expr1 expr2) = (eval expr1) + (eval expr2)
eval (Sub expr2 expr2) = (eval expr1) - (eval expr2)
```

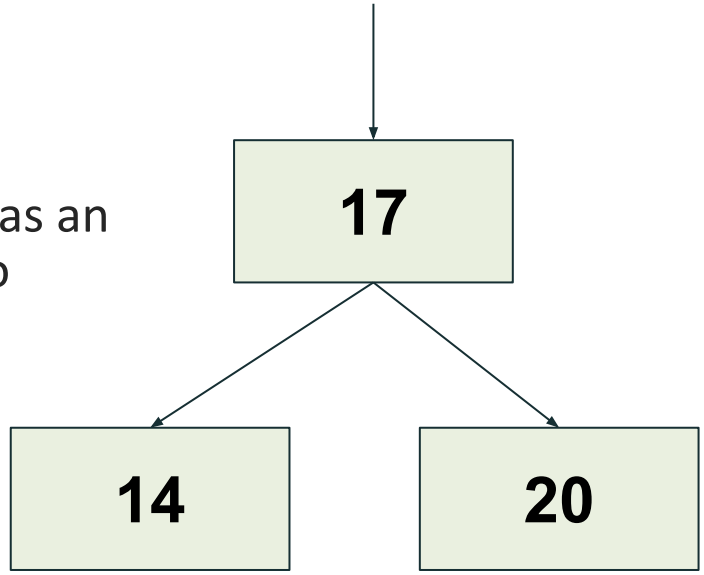
Examples:

```
eval (Lit 0) = 0

eval (Add (Lit 5) (Lit 3)) = (eval (Lit 5)) + (eval (Lit 3)) =
    5 + 3 = 8

eval (Add (Sub (Lit 7) (Lit 3)) (Sub (Lit 3) (Lit 7))) =
    (eval (Sub (Lit 7) (Lit 3))) + (eval (Sub (Lit 3) (Lit 7))) =
    eval (Lit 7) - eval (Lit 3) + eval (Lit 3) - eval (Lit 7) =
    7 - 3 + 3 - 7 = 0
```

Recursive Types

```
data NTree = NilT |
           Node Integer NTree NTree
```

A tree is either empty or it is a node which has an integer label and two branches (that are also trees).



Example:

```
Node 17 (Node 14 NilT NilT) (Node 20 NilT NilT)
```

```
depth :: NTree -> Integer
depth NilT = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)
```

Recursive Types

Finally, the most obvious example of a recursive type is what we've been using the most. Here's one way of defining a list of integer:

```
data ListInteger = EmptyList |
          Cons Integer ListInteger
```

same thing in 'record syntax' looks like this (and is more readable):

```
data ListInteger = EmptyList |
          Cons {head::Integer, tail::ListInteger}
```

Recursive Types

```
data ListInteger = EmptyList |
          Cons Integer ListInteger
```

What if we don't want just a list of Integers, but rather a list of anything? Here's a polymorphic list:

```
data List a = EmptyList |
              Cons a (List a)
```

or

```
data List a = EmptyList |
              Cons {headList::a, tailList::List a}
```

Polymorphic Types

```
data Pairs a = Pr a a

Pr 2 3 :: Pairs Int

Pr “a” “b” :: Pairs [Char]

Pr [] [] :: Pairs [a]

Pr [True] [1,2,3] :: Error. Type Mismatch
```

Polymorphic Types

Binary Trees:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

and some Functions:

```
depth :: Tree a -> Integer
depth Nil = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)

collapse :: Tree a -> [a]
collapse Nil = []
collapse (Node x t1 t2) = collapse t1 ++ [x] ++ collapse t2
```

Polymorphic Types

A Useful Built-in Type:

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

This is a useful type for functions that may need to return null values:

```
errDiv :: Integer -> Integer -> Maybe Integer

errDiv n m
 | m /= 0  = Just (n `div` m)
 | otherwise = Nothing
```

Algebraic Types

You may need to use a sophisticated level of type declaration in your project.

Chapter 14 (Algebraic Types) is a good reference and is where all of the past examples are taken from.

Patterns of Computation

Being made up of functions and types, Haskell offers an enormous capability for generalization of computation through function composition and polymorphism.

For example, let's take a look at these common patterns of computations over lists:

1. Applying to all or Mapping
2. Selecting elements or Filtering
3. Combining the items or Folding

Generalized Patterns: Mapping

```
doubleAll [11,23,13] = [22,46,26]
```

We can implement this function using recursion or list comprehension, like this:

```
doubleAll xs = [2*x | x<- xs]
```

Generalized Patterns: Mapping

```
doubleAll [11,23,13] = [22,46,26]
```

We can implement this function using recursion or list comprehension, like this:

```
doubleAll xs = [2*x | x<- xs]
```

Or we can implement a generalized function that can work for this and many many other similar functions:

```
map f [] = []
map f (x:xs) = [f x | x <- xs]
```

Generalized Patterns: Mapping

```
doubleAll [11,23,13] = [22,46,26]
```

We can implement this function using recursion or list comprehension, like this:

```
doubleAll xs = [2*x | x<- xs]
```

Or we can implement a generalized function that can work for this and many many other similar functions:

```
map f [] = []
map f (x:xs) = [f x | x <- xs]
```

f can be double, triple, sqrt, cube, whatever your heart desires. For double, we can use it like this:

```
map (* 2) [11, 23, 13] = [22,46, 26]
```

Generalized Patterns: Mapping

Let's unpack this:

```
map (* 2) [11, 23, 13] = [22,46, 26]
```

`(* 2)` is two things:

a) It's a 'partially-applied function': it's only applied to one out of its two args. The result is a function waiting to be applied to one other arg. It works just like any function with one arg:

```
(*) :: Integer -> Integer -> Integer
```

```
(* 2) :: Integer -> Integer
```

Generalized Patterns: Mapping

Let's unpack this:

```
map (* 2) [11, 23, 13] = [22,46, 26]
```

`(* 2)` is two things:

b) It's a function that's used as an argument, which makes 'map' a 'higher-order function': "a function that takes a function as an argument or returns a function as a result or both".

```
map :: (a -> b) -> [a] -> [b]
```

```
(* 2) :: Integer -> Integer
```

Generalized Patterns: Mapping

Mapping is *extremely* useful in working with lists. You will see how useful when writing your projects.

Here's another example:

```
convertChars :: [Char] -> [Int]

convertChars xs = map fromEnum xs
```

Or

```
map toUpper "allsmall" = "ALLSMALL"
```

Generalized Patterns: Mapping

Another useful thing you can do in using map, or any higher-order function, is making up functions on the fly:

```
map (\x -> (toUpper (head x)):(tail x))
```

```
    [“mike”, “lisa”] = [“Mike”, “Lisa”]
```

What is this? **(\x ->** `(toUpper (head x)):(tail x)` **)**

Generalized Patterns: Mapping

Another useful thing you can do in using map, or any higher-order function, is making up functions on the fly:

```
map (\x -> (toUpper (head x)):(tail x))
```

```
    [“mike”, “lisa”] = [“Mike”, “Lisa”]
```

What is this? **`( \x ->`** `(toUpper (head x)):(tail x)` **`)`**

backslash **** starts an anonymous (lambda) function

x is its input

-> point to the body of the function

To write it normally, it would be:

```
f x = (toUpper (head x)):(tail x)
```

Generalized Patterns: Filtering

A second useful higher-order function is filter:

```
filter f xs = [x | x <- xs, f x]
```

We've seen examples of this use in list comprehension. The function filter generalizes this usecase for us:

```
filter (< 6) [2,5,3,6,8,9] = [2,5,3]
filter isDigit “abc321” = “321”
filter isSorted [[1,2], [3,1], [8,4]] = [[1,2]]
filter (\x -> sqrt x > x) [2.0,1.5,0.4] = [0.4]

filter :: (a -> Bool) -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
```

Generalized Patterns: Folding

Folding is performing a single operation successively on list items to yield a single result. Many recursive functions over lists, such as sum and product that collapse list values into a single value, can be redefined through folding.
There are 2 types of built-in fold: left-associative and right associative

```
foldl (left associative): [1,2,3,4] -> [(((1,2),3),4)]
foldr (right associative): [1,2,3,4] -> [(1,(2,(3,4)))]
```

Both folds take a function with two input parameters and a list and apply the function to the elements of the list two at a time, one from right to left, the other from left to right.

Generalized Patterns: Folding

```
foldl (left associative): [1,2,3,4] -> [(((1,2),3),4)]
foldr (right associative): [1,2,3,4] -> [(1,(2,(3,4)))]
```

Furthermore, each fold has two versions: one which takes an initial value, one which doesn't. That makes four types of folds total:

```
foldr1 (\x y -> x - y) [1,2,3,4] =  1 - (2 - (3 - 4)) = -2

foldr (\x y -> x - y) 5 [1,2,3,4] = 1 - (2 - (3 - (4 - 5))) = 3

foldl1 (\x y -> x - y) [1,2,3,4] = (((1 - 2) - 3) - 4) = -8

foldl (\x y -> x - y) 5 [1,2,3,4] = ((((5 - 1) - 2) - 3) - 4) = -5
```

the initial value passed to foldl and foldr works exactly as if it was added as an extra element to the head or the end of the input list (respectively).

Generalized Patterns: Folding

Built-in list functions can be redefined in terms of fold:

```
sum list = foldl1 (+) list
product list = foldl1 (*) list
minimum list = foldl1 min list
maximum list = foldl1 max list
concat list = foldl1 (++) list
last list = foldr1 (\x y -> y) [1,2,3]
```

Generalized Patterns: Folding

Built-in list functions can be redefined in terms of fold:

```
sum list = foldl1 (+) list
product list = foldl1 (*) list
minimum list = foldl1 min list
maximum list = foldl1 max list
concat list = foldl1 (++) list
last list = foldr1 (\x y -> y) [1,2,3]
```

You can Read more about these functions and topics in Chapter 10 and 11

Questions?

Questions?

Next topic will be I/O operations from Chapter 8.

I/O: Why is it an issue

Handling Input/Output operations is theoretically challenging in functional languages.

I/O is inherently stateful and creates side effects.

Take these function for instance:

```
getInt :: Integer

inputDiff = getInt - getInt

f n = getInt + n
```

Are they referentially transparent? The value returned can be different at different stages of the execution. If this was allowed this, functions couldn't be statically reasoned about any more. Referential opacity would spill into the entire program

So what does Haskell do?

After years of struggling, the Haskell community finally came up with an idea: Defining a new (polymorphic) type for IO actions:

```
getLine :: IO String

getChar :: IO Char

putStr :: String -> IO ()

putStrLn :: String -> IO ()
```

IO Type

A separate type for IO provides a special environment inside which, and only inside which, referential transparency can be compromised. It prevents the dangerous mixing of the pure and the impure that can taint the whole program.

“they provide a small imperative programming language for writing I/O programs on top of Haskell, without compromising the functional model of Haskell itself”.

Once an IO always an IO.

Primitive I/O actions

```
getLine :: IO String -- reads a line from input and returns it
                     -- as a String (wrapped inside an IO blanket)
getChar :: IO Char -- reads a character from input and returns it
                     -- as a Char (wrapped inside an IO blanket)
```

Primitive I/O actions

```
getLine :: IO String -- reads a line from input and returns it
                     -- as a String (wrapped inside an IO blanket)
getChar :: IO Char -- reads a character from input and returns it
                     -- as a Char (wrapped inside an IO blanket)


putStr:: String -> IO ()
```

() is a tuple type with no elements and has only one possible value: the empty tuple.

This is a convention for all IO functions that return no value (usually print type functions).

Primitive I/O actions

```
putStrLn :: String -> IO ()
putStrLn = putStr . (++ “\n”)
```

Side note: period (.) is a shorthand for function composition:

```
(f . g) x = f (g x)
```

Primitive I/O actions

*Side Note cont'd

Four equivalent ways of writing the same function:

```
putStrLn :: String -> IO ()

1) putStrLn st = putStr (st ++ "\n") -- the simplest way

2) putStrLn = putStr . (++ "\n") -- allows making st implicit

2) putStrLn = putStr . appendLB

   appendLB st = st ++ "\n"  -- no real use in this

3) putStrLn st = putStr $ st ++ "\n"

-- $ changes the associativity of function application. Allows
the parentheses to be dropped. More readable.
```

Primitive I/O actions

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

this is the function ghci calls for printing values of a function you've called at the prompt.

```
ghci> let f x = 2*x
ghci> f 10  --> this will be converted to:
ghci> print $ f 10  or print (f 10)
20
```

Primitive I/O actions

getLine :: IO String

How do you use something that is wrapped in an IO blanket?

How can you access the String read from the input?

Because IO and non IO worlds can't be mixed, you can only access and use that String when you're in IO world. You can't leave the IO world and take it with you. Outside it will be turned into IO String again.

So we'll step into the IO world/context in order to use the String...

do notation: imperative Haskell

The keyword ‘do’ starts a context inside which a sequence of I/O actions, variable binding and everything else can be executed *imperative style*.

Here’s a simple program that reads a value from input, applies a function to it and prints the result:

```
main :: IO ()

main = do putStr “Give me a line: ”
          line <- getLine
          let result = f line
          putStr result
```

do notation: imperative Haskell

let f be the tail function..

```
main :: IO ()
main = do putStr "Give me a line: "
          line <- getLine
          let result = tail line
          putStrLn result
```

```
ghci> main
Give me a line: here's your line
ere's your line
```

do notation: loop

```
applyN :: Integer -> IO ()
applyN n =
    if n <= 0
    then return ()
    else do putStr "Give me a line: "
            line <- getLine
            let result = f line
           putStrLn result
           copyN (n-1)
```

```
return :: a -> IO a -- return wraps the IO blanket of shame around
                    -- values when sending them out the door
```

Questions?

I/O and interactive games

Read the rest of Chapter 8 for more examples on this topic and to better understand the previous examples!