Lec 9 : ILFP COL765

Last lecture

- Data types : Constructors, Destructors, Defining Eqns
  - Even without destructors the structure can be broken down via pattern matching
- Recursive data types : Lists, Trees

- Haskell has static type checking
  - i.e. Every expression has an assigned type
  - 'a' == "a" is a type mismatch Char ≠ [Char]
- Haskell uses type inference (No need to explicity specify Expr types)

Polymorphic type system

- i.e. you can have type variables

Eg: tail has a polymorphic type

$[\alpha] \longrightarrow [\alpha]$

↳ i.e it can take as argument any list and return a value which is a list of the same type

```
: tail [1,2,3,4]
: tail "Subodh"
```

Eg: :t fst

$\forall a, b .\ (a, b) \longrightarrow a$

↳ implicit universal quantification

Type Classes

Motivation

- In many languages such as C++, Java etc. function overloading exists
- Eg: integer comparison, Character comparison, floating pt: comparison
  - In general what we want is an α - Compare, where α is a type variable
    i.e. a variable whose value is a type

- This functionality is introduced in Haskell through type classes

- Eg: Eq, Num, Show type classes

Eq is a type class that provides a (==) operator which has the type $\alpha \rightarrow \alpha \rightarrow$ Bool

- But (==) may not be defined for every type — for types α for which == is defined we can say that α is an instance of Eq class.

Similarly

- Show class have functions which converts values of member types to a string

Eg: Show 5

Type classes can be inherited by using the keyword "deriving"

Eg: data Season = Spring | Summer | ...
deriving (Eq, Show)

Q: data BinT a = EmptyT | Node a (BinT a) (BinT a)

What if we want to use it as a search tree

i.e. values must have a natural ordering → deriving (Eq, Show, Ord)

Function Types

In functional prog. languages, Haskell included, functions are first class objects

- i) they are values, then they must have a type

Lambda Calculus : simple system of representing functions

- **Anonymous functions**

$$[\lambda x.\ \lambda y.\ (2x+y)]$$

- $[\ \ ]$ → Lambda Expression
- λx → Lambda Abstraction
- $(2x+y)$ → body of the function

Eg: Square = \x ⟶ x*x

- (\x ⟶ x.x) 5
- (\x y ⟶ 2x+y) 5 4

Replace every occurrence of var. with the values

[λ calculus theory later]

IO type

:t putStrLn . :t readFile

⟶ what is IO() in the O/P

- Haskell's way of representing that these functions are really not functions

- They are called 'IO actions'
- You can't write a function with type

  IO String ⟶ String

- The only way to combine IO type is to combine them with other functions using the do notation

Eg Suppose you have a function f which takes String ⟶ Int
How can you apply f to the o/p of readFile ?

```
main = do
  s <- readFile "somefile"
  let i = f s
  putStrLn (show i)
```

get the string out of IO Action

] Note no "in" clause
- because we are in a do-block

Defining data types

- Use the keyword `data`

→ Constructors

→ List of functions

Option or Maybe DoTo

Let us consider a simple fnc which searches over a lst for an element satisfying a given pred, p, and returns the first element satisfying p.

→ What should be done if none of the elements satisfy p?

- Raise an error
- Write an explicit check fnc.

- What we are trying to accomplish is a fnc that may or may not succeed!

data Maybe a = Nothing | Just a

Eg:

```
myHead ls =
    case ls of
        []     ⟶ Nothing
        (x:xs) ⟶ Just x
```

Recursive Datatypes

```
data myList a = Nil | Cons a (myList a)

listlength Nil = 0
listlength (Cons x xs) = 1 + listlength xs
```

:

Continue with BinTree recursive def^n.

```
preorder Empty T = Nil
(Node n lT rT) = [N] ++ preorder lT
                     ++ Preorder rT
```

Abstraction in data types

- So far the way we have seen the data types constructed, the structure of the data type is revealed

- Abstract data type def^n hide the internal structure, and access to its components is through its interface

↓

Information Hiding

Features of ADT

- implemented with the type class system or module system in Haskell
- Access to datatype is through methods & operators; internal impl. is hidden (i.e constructors) & no pattern matching is available
- The exact type of methods & operators is exposed in "a signature"
  - → may also expose axioms that the structure must obey!

Eg: Stack via Class

```
Class Stack where
  push  :: a -> Stack a -> Stack a
  top   :: Stack a -> a
  pop   :: Stack a -> Stack a
  Empty :: Stack a -> Bool
```

Signature

Every instance of this type class is a separate implementation

Module

```
data MyStack a
   = MS [a]
```

```
instance MyStack [] where
  push x xs = x : xs
  top (x:xs) = x
  pop (x:xs) = xs
  empty [] = True
  empty (x:xs) = False
```

---

module system (top, push, pop..) → visibility

```
module MyStack, where
```

Constructor ms is not visible ⋮ def".

when we "import MyStack"