Lec7 : ILFP

Higher order functions on lists

- maps
- filters
- fold (l) (r)

→ 3 common fncs applied to Containers

- Map : applies f to each element of the Container
- filter : applies f to filter (exclude) some elements from the container
- fold : another name for → reduce. on computation

$$map\ f\ ls = \begin{cases} [] & \text{if } ls = [] \\ (f\ (head\ ls)) : (map\ f\ (tail\ ls)) & \text{otherwise} \end{cases}$$

where $f : A \longrightarrow B$

$\wedge$ ls is A lst

$$filter\ pred\ ls = \begin{cases} [] & \text{if } ls = [] \\ (head\ ls) : (filter\ pred\ (tail\ ls)) & \text{if } pred\ (head\ ls) \\ filter\ pred\ (tail\ ls) & \text{otherwise} \end{cases}$$

Eg: choosing evens or odds from the list.

pred x = if x mod 2 == 0
then True
else False

positives ls = filter pos ls

where pos x = x > 0

```
foldl f e ls =
        case ls of
          []     ⟹ e
          x : xs → foldl f (f x e) xs
```

initial Element

f is a binary function

$$f : A \times B \longrightarrow B$$

$e : B$ → initial value

a_0

e

a_1

$f(a_0, e)$

$f(a_1, f(a_0, e))$

a_n

$f(a_m, f(a_{n-1}, f(a_{n-2}, \ldots, f(a_0, e)\ldots)))$

```
foldr f e ls =
        case ls of
          []     → e
          x : xs → f x (foldr f e xs)
```

What is remarkable is that map & filter can be defined in terms of foldr

```
map f ls = foldr f' [] ls
           where f' x xs = (f x : xs)

filter p ls = foldr f' [] ls
              where f' x xs = if p x then x:xs
                              else xs
```

Notice that foldl is tail-recursive
and foldr is not

** foldl & foldr will produce the same o/p
so long as f is associative

Use of map & fold (reduce)

- Large scale computations on large datasets
  - Google's MapReduce, Hadoop etc.

In 2008 : John Dean & , Ghemawat attributed functional programming for their proposed abstraction! paper

Datatypes & Structural Induction

Booleans, Integers, Reals, Characters, Strings

↓

for each of these primitive types
a unique type structure exists
in functional languages

In general how
we represent structured data depends on
the following:

1. Constructors : which permit the construction and extension of the structure → the layout of data
i.e. how the data representation is built up

2. Destructors : permit breaking of the structure to its component parts

3. Defining eqn : constructors may be used to reconstruct a data structure

Imp. of datatypes

- provide implicit context for many apps. so that the programmer doesn't have to provide the context
- limit the set of semantically valid operations that can be performed
- if known at compile time, then can be used for optimisations

Compositive types

Tuples, lists

Let us take lists

Name of the type constructor

Data constructors (or just Constructors)

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

Note a recursive constructor

Constructors : Nil, Cons

Destructors : head, tail

Defining Eqn : $l = \text{Nil} \vee l = \text{Cons (head } l\text{) (tail } l\text{)}$

```
data Color = Red | Green | Blue
```

```
data Point a = Pt a a
```

named data constructor

Point → tuple type

binary cartesian product

for any type a, Point a defines a cartesian points that use t as the coordinate type!

type sign of Pt : $a \to a \to \text{Point } a$

Other examples

Option Datatype ⟶ a polymorphic datatype

data Maybe t = Nothing | Just t

Constructors : Nothing , Just

Destructors : show

Defining Eqn : M = Nothing ∨ M

= Just (show M)

Binary Tree Type

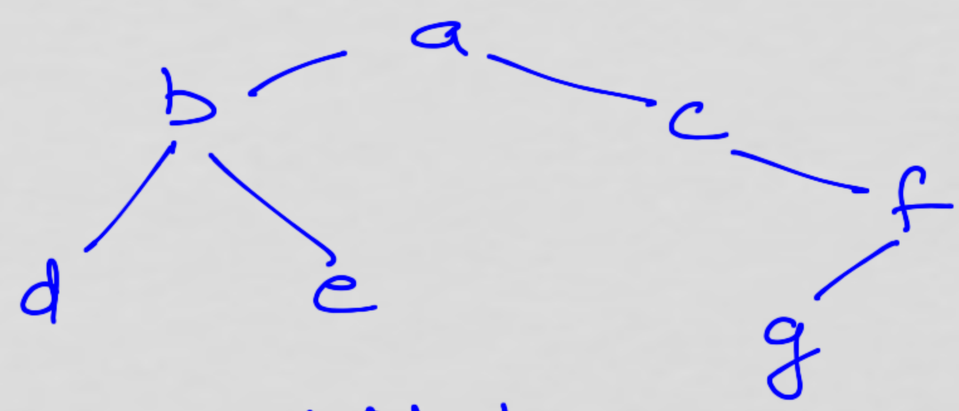
```
data BinTree a = Empty | Node a (BinTree a)
                                (BinTree a)
```

Polymorphic bintree whose elements are either leaf nodes containing a value of type a or internal nodes containing two subtrees

Constructors : Empty , Node

Destructors : root, left subtree, right subtree

Defining Eqn : T = Empty V T = Node (root T)
(leftSubTree T)
(rightSubTree T)



t1 = Node a (Node - - - - - -)