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

$e \searrow$

a_1

$f(a_0, e) \searrow$

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

... a_n

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

```
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

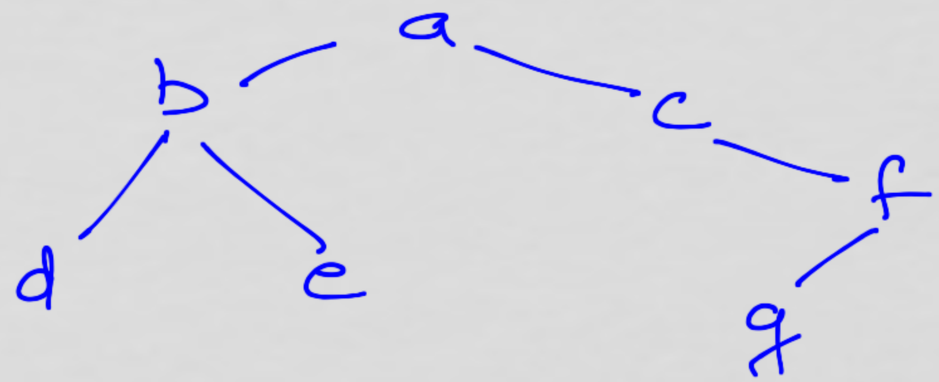
```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
                 deriving (Show)
```

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 b (Node d Empty Empty)
(Node e Empty Empty))
(Node c Empty (Node f (Node g Empty
Empty))

Indeed foldr f e ls and foldl f e (reverse ls) are 'almost equal'.

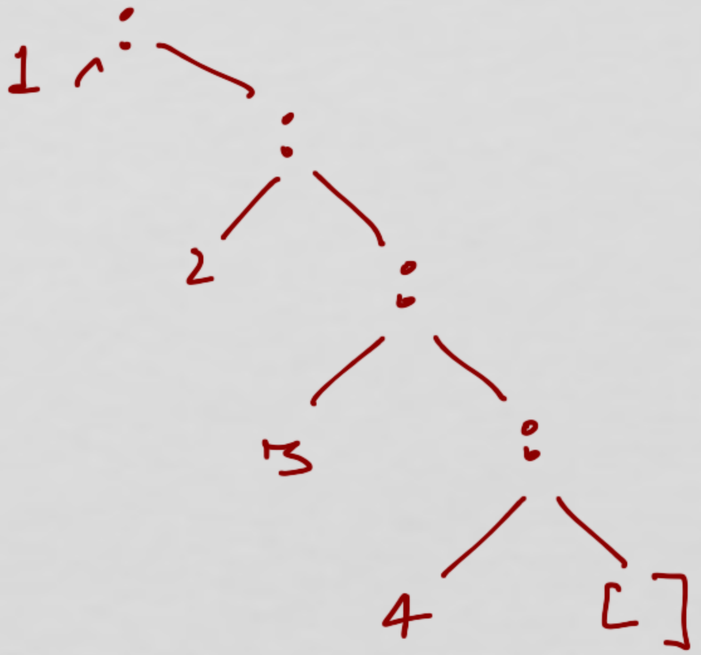
- And no associativity has no role to play there

- But for lists with malformed values or infinite lists some differences may crop up!

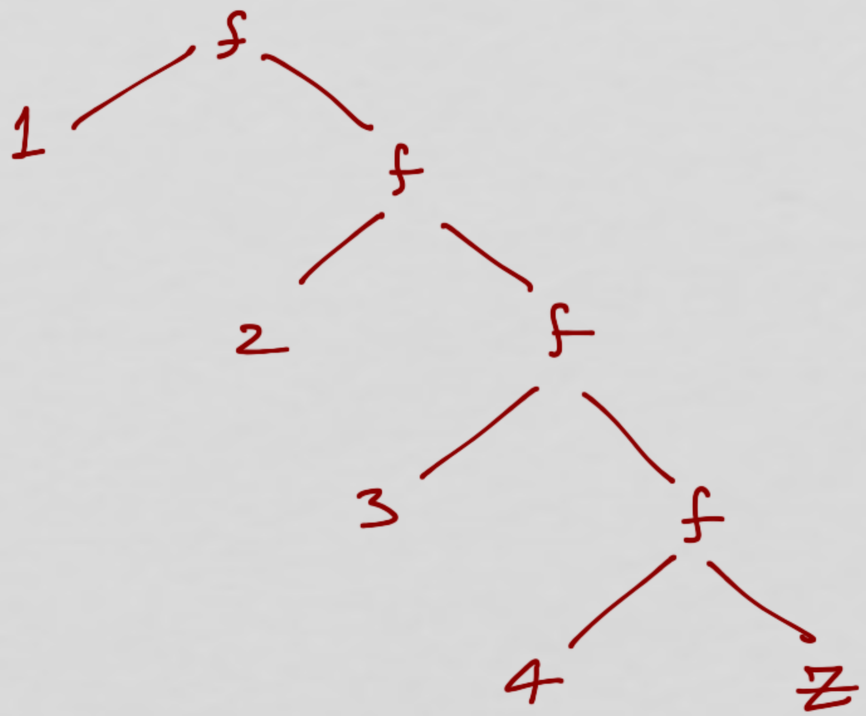
— Note reverse version indeed requires us to evaluate over the entire list before any results are produced

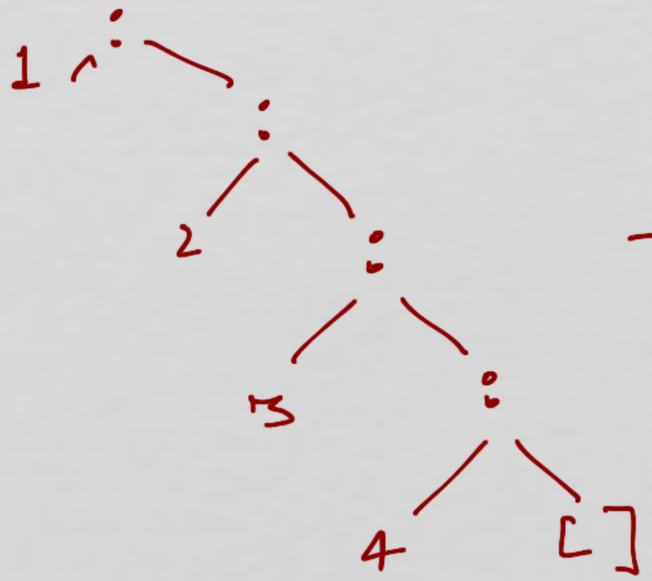
- thus it wouldn't work [with foldl rev ls] on infinite lists/structures, while foldr can give you partial evaluations.

foldr

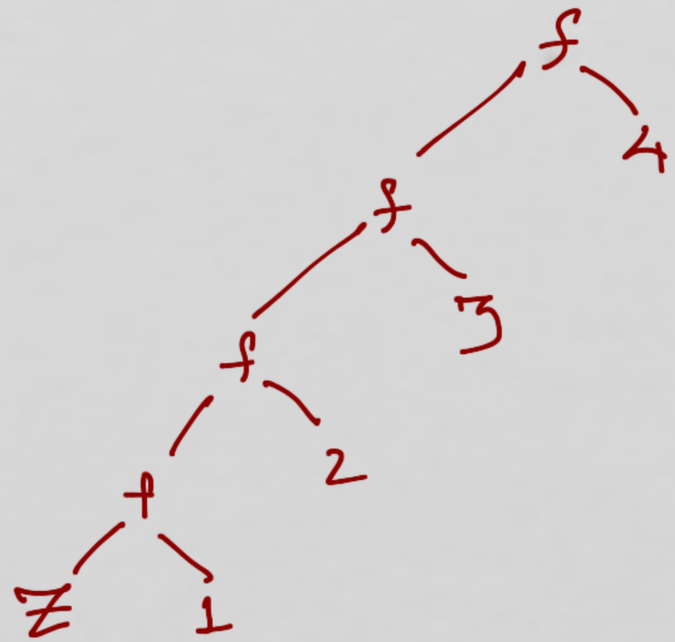


foldr f z →





foldl f e →



Structural Induction

- Let U be a set called universe
- Let $B \subseteq U$ be a non-empty set called the basis
- Let K be the constructor set s.t.

$$\forall f \in K, \quad \underbrace{\alpha}(f) \geq 0$$

↳ arity of f

- Then $f : U^n \longrightarrow U$

- Let $\mathcal{X}$ be family of subsets of U Soto $\forall X \in \mathcal{X}$ satisfies the following Condition

  - Basis: $B \subseteq X$
  - I.S.: if $f \in K$, $\alpha(f) = n \geq 0$ and $a_1, a_2, \ldots, a_n \in X$ then $f(a_1, a_2, \ldots a_n) \in X$

- Set A is inductively defined from B, K, U if it the smallest set satisfying

$$A = \bigcap_{X \in \mathcal{X}} X$$

A is said to have been generated from the basis B and the rules of generation

$f \in K$

Lemma 1: $A \in \mathcal{F}$ if A and $\mathcal{F}$ are as in the above defⁿ.

Proof: $B \subseteq X$, for each $X \in \mathcal{F}$

we have $B \subseteq A = \bigcap_{X \in \mathcal{F}} X$

$\Rightarrow$ A satisfies the basis condition

Consider $f \in K$ & $a_1, \ldots a_{\alpha(f)} \in A$.

By defⁿ. $a_1, a_2, \ldots, a_{\alpha(f)} \in X \; \forall X \in \mathcal{F}$

$\therefore f(a_1, \ldots, a_{\alpha(f)}) \in X \; \forall x \in \mathcal{F}$

$\Rightarrow f(a_1, \ldots, a_{\alpha(f)}) \in A$

Hence $A \in \mathcal{F}$

Rules of generation

- aritExprs generated from $\mathbb{N}$
- Basis: $\forall n \in \mathbb{N}$, n is an aritExpr
- I.S.: $e, e' \in$ aritExpr
  $\Rightarrow$ add (e, e'),
  mult $(e, e') \in$ aritExpr.

add $(-,-)$, mult $(-,-) \in \mathbb{K}$

In the BNF

$e, e' := n \in \mathbb{N} \mid \text{add}(e, e') \mid \text{mult}(e, e')$

Principle of structural induction

- Let $A \subseteq U$ be inductively defined by

  $B \neq \phi$ and $K \neq \phi$

- A property P holds on A provided
  - P is true for all basis elements
  - $\forall f \in K$, if P holds for $a_1, \ldots a_{\alpha(f)} \in A$

    $\Rightarrow$ P holds for $f(a_1, \ldots a_{\alpha(f)})$

Eg:

$$e, e' := n \in \mathbb{N} \mid (e + e') \mid (e * e') \mid (e - e')$$

Here

$$U = \left(B \cup \{(,), +, *, -\}\right)^*$$

$$B = \mathbb{N}$$

$$K = \{f_+, f_*, f_-\}$$

- Let e be some expr generated by this BNF
- Let e' be prefix of e
  - ↳ may not be an expr. of the language

Let $\ell(e')$, $R(e')$ denote respectively the # of (&) in e'.

Let $P(e)$ = for every prefix e' of e
$\ell(e') \geq R(e')$

Proof?

By structural induction

Basis: $\forall n \in \mathbb{N}$, $P(n)$ holds ∵ no natural number $\ell(n) = R(n) = 0$

I.H.: for any e of the form $\{e\}$ $\odot \in \{+, *, -\}$, $\{e\}$ $\{e\}$ are expressions in the language

$P(f)$ holds, ie $\forall f'$ prefix of f

$$\mathcal{L}(f') \geq R(f')$$

& $P(g)$ holds, (similiar to above reasoning)

I.S.: Exhaustive case analysis for e

---

prefix: $e' = \epsilon$. $\mathcal{L}(\epsilon) = R(\epsilon)$

. $e' = ($. $\mathcal{L}(() \geq R(()$

$1 > 0$

. $e' = (f'$. $\mathcal{L}(e') = 1 + \mathcal{L}(f')$

$\Rightarrow R(f')$ (By I.H.)

$> R(e')$

$e' = (f' \odot$

$e' = (f \odot g'$

$e' = (f \odot g$

$e' = (f \odot g)$

More Eg:

data nat = Z | S nat | P nat

1-ary constructors