COURSE NAME: CS558

COURSE DATE: 9/11/18

**Review of last time:**

Datatype

data MyList a = MyEmpty

| MyCons a (My List a)

MyCons 1 (MyCons 1 MyEmpty) : MyList Int

(2: (1:[ ] )) : : [ Int]

You can declare things with multiple type instructors, polymorphic, etc.

data Maybe a = Nothing

| Just a

This lets you model when you may not actually have a value.

For example, map a function and increment the numbers by 1:

xs = [Just 1, Just 2, Nothing, Just 4] : : [Maybe Int]

map (\x -> n +1) xs -🡪 this fails to typecheck!!! The \x is Int 🡪 Int but the list contains clause of type Maybe Int, even if the Nothing is not there.

**Binary trees**

For example, we are interested in a tree with value 1 and then a level of sub-values (value2, value3).

data Tree a = Leaf a

1 Node (Tree a) (Tree a)

treeMap: : (a->b) -> (Tree a) -> (Tree b)

treeMap f (leaf x) = Leaf (f x)

treeMap f (Node t1 t2) = Node (treeMap f t1) (treeMap f t2)

treeMap (\n -> n ‘mod’ 2 = = 0) t

Node (node (Leaf False) (Leaf Tree))

(Leaf False)

**Foldr and Lists**

foldr f v (1 (2 (3 [ ] )))

f 1 (f 2( f 3 v ))

Node (Node (Leaf 1 ) ( Leaf 2)) (Leaf 3)

The empty list has no values stored in it so it usually becomes a constant. But our base case does have a value stored, so when we turn it into a similar type of expression, it will look differently:

Replace Leaf 3 with some function such as (fLeaf 1) (fLeaf 2)

Also, replace the Nodes with a function called fNode which aggregates the values that come out of the recursive calls of the two sub trees:

fNode (fNode (fLeaf 1) (fLeaf 2)) (fLeaf 3)

treeFold : : ( a->b) -> (b -> b -> b

treeFold fLeaf fNode (leaf x) = fLeaf x

treeFold fLeaf fNode (Node t1 t2)

= fNode (treeFld fLeaf fNode t1) (treeFold fLeaf fNode t2)

To be more concise:

= treeFold id (+) t

Write as:

treeFold id (+) (Node (Node (Leaf 1) (Leaf 2) ) (Leaf 3)

= (treeFold id (+) (Node (Leaf 1) (Leaf 2) + (treeFold id (+) (Leaf 3))

= (( treeFold id (+) (Leaf 1) + (treeFold id (+) (Leaf 2))) + (treeFold id (+) (Leaf 3))

= ((id 1) + (id 2)) + (id 3) = (1 +2) +3 = 6

So you an aggregate over trees just as you can over lists.

You can implement a tree map using a tree fold:

treeMap f t = treeFold ( \n -> Leaf (fn))

We can prove things about functions on trees just like on Lists.

For Lists, we have a structural induction principle:

To prove all Lists P(L), prove P ( [ ] ) and prove all x and all xs.

For binary trees, you get a similar induction principle:

dataTree a = Leaf a

| Node (Tree a) (Tree a)

To prove All L :: Tree a. P (t), then prove All x :: a. P (Leaf x) and also prove All t1: : Tree a. and All t2: Tree a.

P (t1) and P (t2), then this implies P (Node t1 t 2)

Example of this:

Proving correctness of treeSum in tree Fold:

treeSum : : Tree Int -> Int

treeSum (Leaf n) = n

treeSum (Node t2 t2) = (treeSum t1) + (treeSum t2)

Prove: All t .. Tree Int . tree Sum t = treeFold id (+) t

To prove the base case: treeFold id (+) (Leaf n ) = id n = n

treeSum (Leaf n ) = a

Prove Inductive case, t – Node t1 t2

Assume treeSum t1 = treeFold id (+) t1 and treeSum t2 = treeFold id (+) t2

= (treeFold id (+) t1) + (treeFold id (+) t2)

Then treeFold id (+) (Node t1 t2)

Finish by proving treeMap and treeFold:

All f: : a -> b All L: : Tree a

treeMap f t = treeFold (\n -> Leaf (f n )) Node f

First, prove the base case:

treeFold (\n -> Leaf (f n )) Node (Leaf x)

= (\n -> Leaf (f n )) x = Leaf (f x)

treeMap f (Leaf x) = Leaf (f x)

Second, prove inductive case:

t = Node t1 + t2

Assume treeMap f t1 = treeFold (\n + Leaf (f n ) ) Node t1

Tree Map f t2 = treeFold (\n -> Leaf ( f n )) Node t2

TreeFold (\n -> Leaf (f n )) Node (Node t1 + t2)

= Node (treeFold (\n -> Leaf (f n)) Node t1) (treeFold (\n -> Leaf (fn)) Node t2)

= Node ( treeMap f t1) (treeMap f t2)

= treeMap f (Node t2 t2)

##end notes##