Class: CS 558 9/6/18

Lecture

* HW will be graded and returned soon.
* Assignment 2 will be posted soon.

We are finishing Chs. 4, 5 and will start Ch. 6 content soon.

Last time: Prove by induction on the structure of Lists

$\forall L.\ P(L)$

L: List, P: property

① Prove base case with List (empty or non-empty)

$P([\,])$

② Inductive ~~base~~ case:
How can we prove for all NON empty Lists?

$\forall x.\ \forall xs.\ P(xs) \Rightarrow P(x :: xs)$

x: element, xs: List

So we assume it works for all since we tacked the ~~non-empty~~ element at the beginning.

Haskell Datatypes

Lots of types of polymorphic type:

[Int] [Bool] [[Int]] etc.

We can use this in type declaration of functions, such as:

length : [a] → Int.

map : (a → b) → [a] → [b]

This is valid for any uniform instantiation of type variables.

List types:

- Empty []
- Non empty X : XS

a ↙ [a] ↘

} This is a recursive definition b/c the non-empty list contains itself.

Variant types (sum):

Can have different things in it.

Define them by the data keyboard!!

Example: Data type for natural numbers: (0, 1, 2...)

data Nat = zero | Succ Nat

name of data type ↘ type constructors ↙ type argument (to suc)

Just as you have reg. functions, you can do similar things w/ Haskell:

0 ~~ represented by ~~> zero

1 ~~ represented by ~~> Suc Zero : Nat

Succ 1 fails b/c Succ 1 : Int not Nat.

Succ : Nat → Nat.

Say you have a list and want to add One to them:

```
map Succ [zero, Succ zero]
    [Succ zero, Succ (Succ zero)]
```

What if you wanted to add Nat together?

```
add Nat : Nat → Nat
add Nat  zero ~2 = ~2
add Nat (Succ ~1) ~2 = Succ (add Nat ~1 ~2)
```

What about mathematical induction?

Use to prove for ∀n : Nat. P(n),

① P(zero)

② ∀~ : Nat. P(n) ⇒ P(Succ ~)

Another use: Representing shapes!!

```
data shape = Circle Double
           | Square Double
           | Rectangle Double Double
           | EqTri Doube   (equilateral Δ)
```

What function can calculate area of a shape?

```
area : Shape → Double
area (Circle r) = pi * r * r
area (Square n) = n * n
area (Rectangle m n) = m * n
area (EqTri n) = 0.5 * n * h  where h = 0.5 * n * (sqrt 3)   ③
```

What if we added Pentagon?
We have to add it to type declaration and to <u>every function</u>! Otherwise it won't be recognized.

Another type class: Show
It a class of types { show .. a → String
This function is used to produce the String.

Default → autogenerates show : Shape → String.
<u>OR</u> you can promote your own orientation:
show (Circle r) .....

((Circle 1.0) == (Circle (2.0 - 1.0)) } Fouls to type check!

Instancy Eq Shape where:

Auto generating == will fail if there is a function type as adjustment to a type constructor.
Using a List example (to redefine number type):

```
data MyList a = MyEmpty
              | MyCons a (MyList a)
myMap :: (a->b) -> MyList a ->
                   MyList a
```

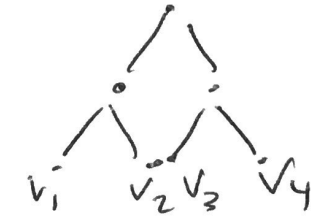
} MyList a is polymorphic. It's also recursive b/c defined by itself.

```
myMap f MyEmpty = MyEmpty
myMap f (MyCons x xs) = MyCons (f x)
                        (MyMap f xs)
```

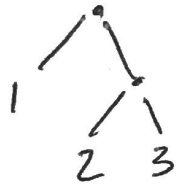
a → myList 'a'

What if we wanted a binary tree?

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



Polymorphic Type




```
=> Node (Leaf 1)
     (Node (Leaf 2) (Leaf 3))
     : Tree Int
```

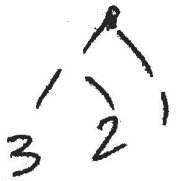
```
treeMap : (a→b) → (Tree a) → (Tree b)
treeMap f (Leaf x) = Leaf (f x)
```

What if you wanted to flip?

```
treeFlip : Tree a → Tree a
treeFlip (Leaf x) = Leaf x
treeFlip (Node t1 t2) = (treeFlip t2)
                        (treeFlip t1)
```

} So leave it alone!

Here, you can't just flip the order.



What about a fold? Iterate across data structure:

```
treeFold : (a->b) -> (b->a->b) -> (Tree a) -> b
treeFold fLeaf fNode (Leaf x) = fLeaf x
treeFold fLeaf fNode (Node t1 t2)
    = fNode (treeFold fLeaf fNode t1)
            (treeFold fLeaf fNode t2)
```

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

~~Turns into~~ Turns into:

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

Could implement as:

```
sumTree Tree Int -> INT
sumTree t = treeFold (\n->n) (+) t
```