CS4450 Lecture 15 Notes

Friday, September 21, 2018 2:59 PM

Today, we'll continue to talk about high-order programming.

Any Haskell list can be viewed as a "tree" (shown in page 31 of the slides)

Let's try and figure out iadd using foldr. We'll use foldr to implement iadd.

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v () = v
foldr f v (x:xs) = x 'f' (foldr f v xs)

How did we figure out the type of **f**? It's a function with two arguments, and it's really only used if there's a list as an argument. **f** is applied to x, so the first argument must have the type of x. What type is x? **a**. Therefore it starts with (a -> ->). It's second argument is the type of (foldr f v xs), which is type **b**. It returns a **b** as well.

What we just computed is called the "most general definition", which is a technical term (this will not be on the midterm, but will probably be on the final).

Let's try and figure out map using foldr.

foldr :: (a -> b -> b) -> b -> [a] -> b

What's the type of map?

map :: (c -> d) -> [c] -> [d]

The code for map will be posted on Canvas. I also have it here:

```
map :: (c -> d) -> [c] -> [d]
map g cs = foldr (\c ds -> g c : ds) [] cs

filter :: (c -> Bool) -> [c] -> [c]
filter p cs = foldr f v cs
```

New syntax: \ c ds -> e -- This is a nameless function that is equivalent to **\c -> \ds -> e**

The function **foldr** is useful because it makes some recursive processes easier.

Note: Accumulator passing style is a "fold left" style. **foldl** "folds from the left"

Why two folds? Sometimes using one can be more efficient than the other. Two recursion patterns => two folds.