CS 350 2024-25 Semester I

Satyadev Nandakumar

August 9, 2024

Outline

Lecture 4 outline

- Type of map, and filter in detail
- foldr
- composition
- Using higher order functions
  - Aggregation
  - List of primes
  - word puzzle

Type of `map`

```
map f []     = []
map f (x:xr) = (f x) : (map f xr)
```

Suppose `xs::[a]`.

Then `f` should take as input, an element of type `a`.

Type of `map`

```
map f []     = []
map f (x:xr) = (f x) : (map f xr)
```

Suppose `xs::[a]`.

Then `f` should take as input, an element of type `a`.

There is no constraint on the output of `f`.

So `f::a -> b`.

Type of `map`

```
map f []     = []
map f (x:xr) = (f x) : (map f xr)
```

Suppose `xs::[a]`.
Then `f` should take as input, an element of type `a`.
There is no constraint on the output of `f`.
So `f::a -> b`.
Hence the output of `map` will be a list of type `[b]`.

Type of `map`

```
map f []     = []
map f (x:xr) = (f x) : (map f xr)
```

Suppose `xs::[a]`.
Then `f` should take as input, an element of type a.
There is no constraint on the output of `f`.
So `f::a -> b`.
Hence the output of `map` will be a list of type `[b]`.

```
map :: (a->b) -> [a] -> [b]
```

Type of `filter`

Filter returns only those elements in a list which satisfy a boolean function (i.e. a function that returns `Bool` type).

```
filter pred [] = []
filter pred (x:xr) = if (pred x) then (x:(filter pre
```

Suppose `xs::[a]`.

Type of `filter`

Filter returns only those elements in a list which satisfy a boolean function (i.e. a function that returns `Bool` type).

```
filter pred [] = []
filter pred (x:xr) = if (pred x) then (x:(filter pre
```

Suppose `xs::[a]`.
Then `pred :: a -> Bool`.

Type of `filter`

Filter returns only those elements in a list which satisfy a boolean function (i.e. a function that returns `Bool` type).

```
filter pred [] = []
filter pred (x:xr) = if (pred x) then (x:(filter pre
```

Suppose `xs::[a]`.
Then `pred :: a -> Bool`.
Since filter should filter a subset of `xs`, the output list must have type `[a]`.

Type of `filter`

Filter returns only those elements in a list which satisfy a boolean function (i.e. a function that returns `Bool` type).

```
filter pred [] = []
filter pred (x:xr) = if (pred x) then (x:(filter pre
```

Suppose `xs::[a]`.
Then `pred :: a -> Bool`.
Since filter should filter a subset of `xs`, the output list must have type `[a]`.
Thus, we have

```
filter :: (a->Bool) -> [a] -> [a]
```

The `foldr` function.

Consider the task of summing elements of a list.
We start with an initial value of 0,
and keep adding the elements of the list in sequence. We could write

```
addList [] = 0
addList (x:xr) = x+(addList xr)
```

The `foldr` function.

Consider the task of summing elements of a list.
We start with an initial value of 0,
and keep adding the elements of the list in sequence. We could write

```
addList []     = 0
addList (x:xr) = x+(addList xr)
```

How about product?

The `foldr` function.

Consider the task of summing elements of a list.
We start with an initial value of 0,
and keep adding the elements of the list in sequence. We could write

```
addList [] = 0
addList (x:xr) = x+(addList xr)
```

How about product?

```
prodList [] = 1
prodList (x:xr) = x*(prodList xr)
```

The `foldr` function.

Consider the task of summing elements of a list.
We start with an initial value of 0,
and keep adding the elements of the list in sequence. We could write

```
addList [] = 0
addList (x:xr) = x+(addList xr)
```

How about product?

```
prodList [] = 1
prodList (x:xr) = x*(prodList xr)
```

These codes look very similar. The only difference is that 0 is replaced with 1, and + with $*$.
Can we generalize this?

Deriving `foldr`

The components are:

1. a value for the empty list
2. a binary function (+ or * in the previous examples)
3. a list of elements

`addList [1,2,3]` operates as (1+(2+(3+0)))
`prodList [10,20,30]` operates as (10*(20*(30*1)))
Abstracting,

Deriving `foldr`

The components are:

1. a value for the empty list
2. a binary function (+ or * in the previous examples)
3. a list of elements

`addList [1,2,3]` operates as (1+(2+(3+0)))
`prodList [10,20,30]` operates as (10*(20*(30*1)))
Abstracting,

```
foldr op z []       = z
foldr op z (x:xr)   = x `op` (foldr op z xr)
```

Deriving `foldr`

The components are:

1. a value for the empty list
2. a binary function (+ or * in the previous examples)
3. a list of elements

`addList [1,2,3]` operates as (1+(2+(3+0)))
`prodList [10,20,30]` operates as (10*(20*(30*1)))
Abstracting,

```
foldr op z []       = z
foldr op z (x:xr)   = x `op` (foldr op z xr)
```

Remember that 2 argument function op can be applied to arguments as `op x y` or `x `op` y`.

Type of `foldr`

Let `xs :: [a]`.
Let `z :: b`.
Then `op:: a -> b -> b`.

Type of `foldr`

Let `xs :: [a]`.
Let `z :: b`.
Then `op:: a -> b -> b`.
(Why is the return type `b`? Because that will be the second argument to the upper level `op`.)
Thus we have

```
filter :: (a->b->b) -> b -> [a] -> b
```

Programming using higher-order functions.

Big tip: think of functions as changing lists to lists, as far as possible. We will see examples.

List of primes

Base 60

Why did Babylonians use base 60? (sexagesimal)

A word puzzle

A game: given a word like "food", see whether you convert it into another word, like "soul", using five transition words, where each word differs from the immediately previous one by exactly one word. For example, changing "Foot" into "Ball"

1. Foot
2. Food
3. Fold
4. Bold
5. Bald
6. Ball

Write a Haskell program, that, given two words each having length 4, outputs a sequence of five transition words if a sequence exists, and otherwise outputs the empty list.