

# Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

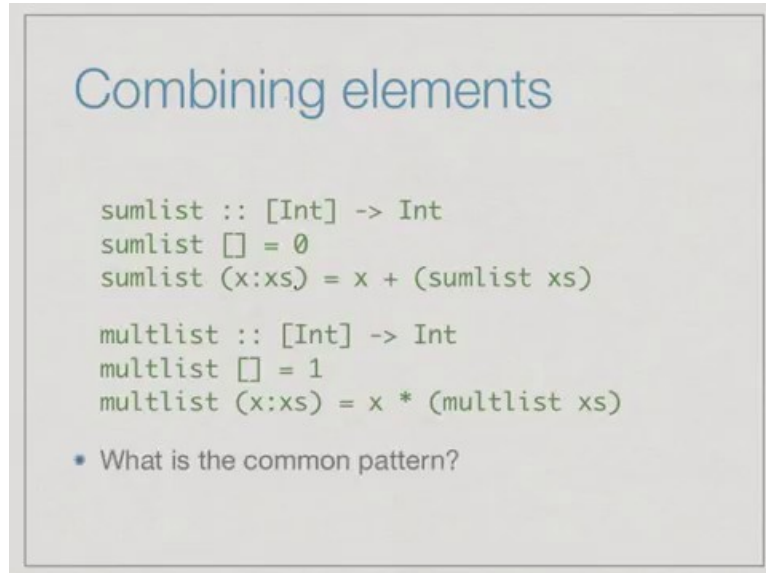
Module # 03

Lecture – 05

Folding through a List

Map and filter are two important higher order functions on list that allows to manipulate entire list at a time. This today, we will look at another type of function, which involves folding a function through a list or combining the lists.

(Refer Slide Time: 00:17)



Combining elements

```
sumlist :: [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + (sumlist xs)

multlist :: [Int] -> Int
multlist [] = 1
multlist (x:xs) = x * (multlist xs)
```

- What is the common pattern?

So, here is an example of combining the elements of list, when we want to take all the values in the list and add them up. So, this is the built in function sum. So, let us call it sumlist. So, that we do not confuse it with the built in function sum. So, if we have an empty list, this sum is 0, if we have a non-empty list, we add the first element to the sum of the rest.

So, this can be also generalized to multiplying the values in a list. So, let us decide that the value of an empty list is 1. And now, if we want to multiply the values of list, we basically want to multiply x1 by x2 and so on. So, we take the head of the list and multiply it with the product of the remaining elements in the list.

(Refer Slide Time: 01:03)

### Combining elements ...

```
combine f v [] = v  
combine f v (x:xs) = f x (combine f v xs)
```

So, the common pattern is that we are combining a function across a list with an initial value. So, if we have a function, so in this case the functions were hardwired as sum or multiplier, but in general, we have function  $f$ . So, if we have the empty list, then we have this initial value  $v$  and if not, then what we do is, we apply the function to the current value and then recursively compute the value that we got from the rest of the list.

(Refer Slide Time: 01:35)

### Combining elements ...

```
combine f v [] = v  
combine f v (x:xs) = f x (combine f v xs)
```

- We can then write

```
sumlist l = combine (+) 0 l  
multlist l = combine (*) 1 l
```

So, this is then the way, we can write our earlier functions.

```
sumlist l = combine (+) 0 l
```

```
multlist l = combine (*) 1 l
```

So, when we are doing `sumlist`, the function is plus (+) and when we are doing multiplying list, the function is times (\*). So, the initial value for `sumlist` is 0, the initial value for multiplying list is 1. So, when we combine 0 as an initial value with the function plus, we get the sum of the numbers. When we combine multiply with the initial value as 1, we get the product of the list.

(Refer Slide Time: 02:02).

**foldr**

- The built-in version of `combine` is called `foldr`

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

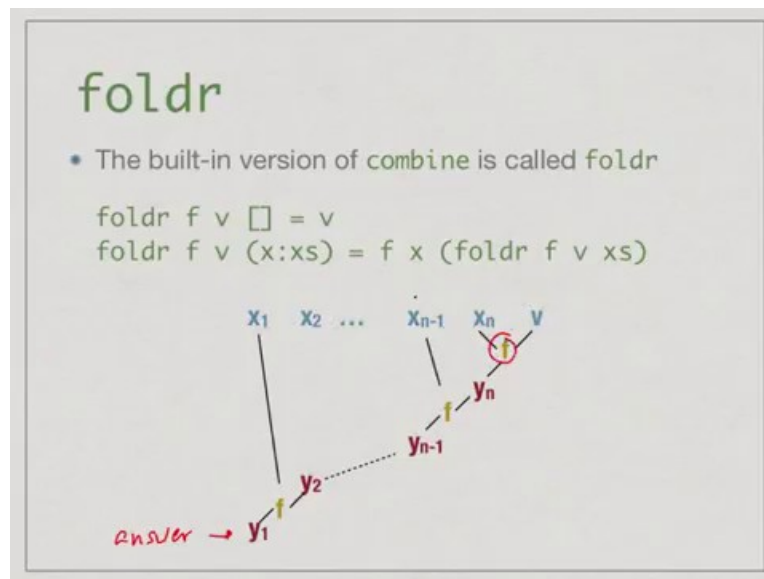
Diagram illustrating the recursive application of `foldr` on a list  $x_1, x_2, \dots, x_{n-1}, x_n$  and an initial value  $v$ :

- The list is represented as  $x_1 \ x_2 \ \dots \ x_{n-1} \ x_n$ .
- The initial value  $v$  is shown in a red circle below  $x_n$ .
- Red annotations show the recursive steps:
  - $f \ x_n \ []$  (above  $x_n$ )
  - $f \ v \ []$  (above  $v$ )
  - Arrows indicate the flow of computation from the end of the list towards the beginning.

So, the function in Haskell which does this is called `foldr`; and this process is called folding the function through list. So, `foldr` is basically just the redefinition of what we wrote for `combine`. So, we take a function and an initial value. For the empty list, we return the initial value and otherwise, we inductively apply the function to the current value, the head of the list and the value, we have computed in the tail of list.

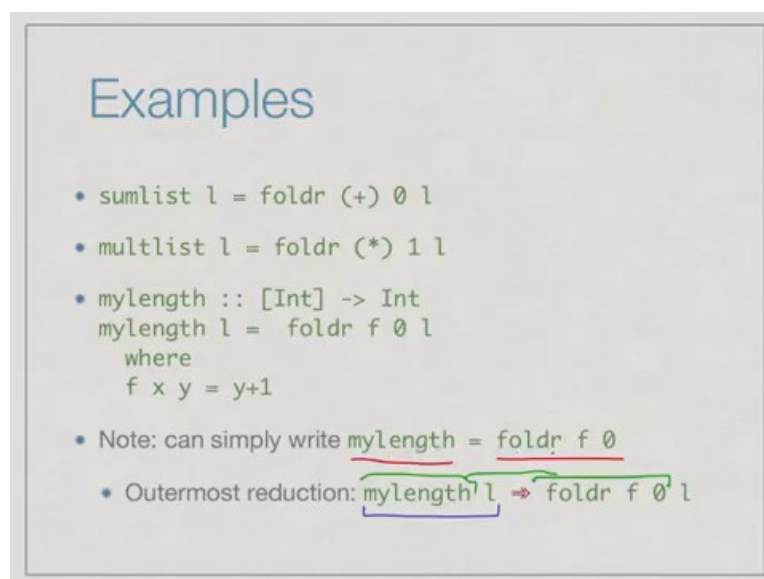
So, we can visualize this as follows, we start with the list  $x_1$  to  $x_n$  and the value  $v$ , which is our initial value. So, now, initially if we just look at the list, then as we inductively go through this, the thing that we do is, we operate on the last value. So, if we have  $x_n : []$ , then this will apply  $f$  to  $v$  and the empty list, it gives us  $v$  and then, apply  $f$  to this value and this value.

(Refer Slide Time: 02:55)



So, the first thing we get is that, we combine  $x_n$  with the initial value  $v$  and we get a new value  $y_n$ . Then, in the previous step, we apply  $x_{n-1}$  to the result of this value and we get a value  $y_{n-1}$ . So, in this way working backwards from the right, so this  $r$  in `foldr` refers to the right. So, working right to left, we keep going. So, at some point, we will combine  $x_2$  with the value  $x_3$  onwards to get  $y_2$  and in the final step, we combine  $x_1$  with that value to get  $y_1$  and this is our final answer. So, this is the pictorial representation of what it means, to fold this function  $f$  from the right through this list with an initial value  $v$ .

(Refer Slide Time: 03:40)



So, therefore, `sumlist`, the function that we wrote, this is same as folding the function plus from the right, using the initial value 0. `Multlist` is the result of folding the function times

from the right with an initial value 1. We can even define the length of a list by an appropriate function, so if we just take the function which starts off with 0 and adds 1 for every value it sees, then we can take that function and fold it.

So, for the empty list, we get 0, for any non-empty list, we get 1 plus result of the function on the remaining thing, because  $f$  of  $x$   $y$  is  $y$  plus 1. So, whatever value, in this case,  $y$  will be the result that has come from the remaining list and we will get  $x + 1$ . So, `mylength`, the length of the list can also be expressed using fold.

So at this point, let us introduce the convention which is often used by Haskell programs. Notice that, we can write `mylength` directly to be this expression `fold f 0`; that is because, when we apply `mylength` to any list, we apply this expression to the list. Then, if I have an expression of this form, remember that Haskell works by outermost reduction, to rewrite the outermost definition. So, it will not try to expand `l`, it will try to expand `mylength`.

So, it will take this expression `mylength` and it will be expanded as `foldr f 0`. So, by using this succinct definition as long as the arguments in the rewritten expression come in the same position, they do not have to be inserted inside. We can omit the argument, when you write these functions. So, this is just the usual piece of notation that when we have such situation, where the arguments are going to be applied with the expression in the same order with no reshuffling, then, we can just use the expression itself as the definition of the function without preferring to graph.

(Refer Slide Time: 05:42)

*foldr appendright [] [1,2,3]*

### Examples ...

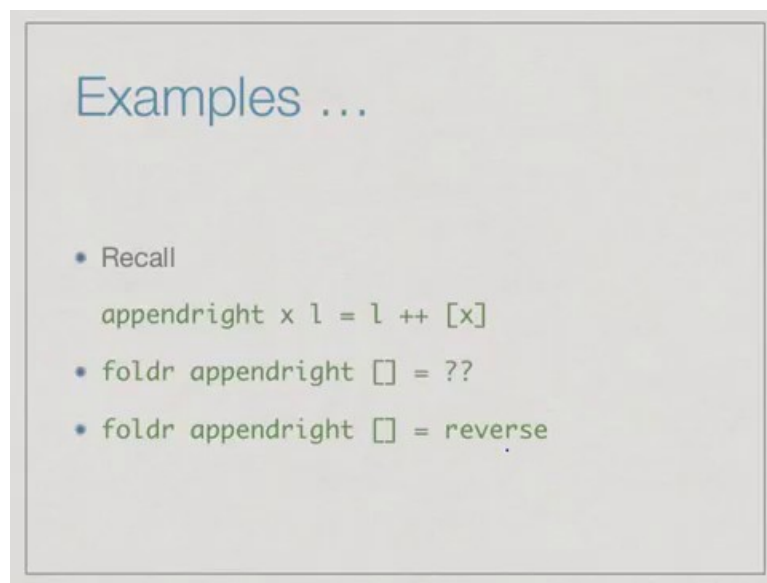
- Recall
  - `appendright x l = l ++ [x]`
- `foldr appendright [] = ??`

The diagram illustrates the foldr operation for `appendright`. It shows the list `[1, 2, 3]` and the initial value `[]`. Red arrows indicate the folding process from right to left. The first arrow points from 3 to `[]`, resulting in `[3]`. The second arrow points from 2 to `[3]`, resulting in `[2, 3]`. The third arrow points from 1 to `[2, 3]`, resulting in `[1, 2, 3]`.

So, remember that one of the early functions we wrote in Haskell was to add a value to the end of the list, instead of instead of appending at the beginning as is done when we do `x:l` we want it to take `l` and stick `x` at the end. So, we can write this for instance as `l ++ [x]`, where `[x]` is the singleton list `x` or we can write an inductive definition, but this is what we need to appendright. So, what would happen if we used `appendright` on the list with initial value empty.

So, lets try to execute it and supposing I apply this `foldr appendright [] [1,2,3]`. So, we have the empty list `[]` and we have `[1, 2, 3]`. So, the first thing is that, I will take this element 3 and combine it to this `[]` using `appendright`. So, this will give me the element on the right `[3]`. Now, we will take this element 2, stick it to the right of this `[3,2]` and you will take this element 1, stick it to the right `[3,2,1]`. So, we can see is that `append r` , `appendright` is shifting each element to the end of the list already created to it is right.

(Refer Slide Time: 07:03)



Examples ...

- Recall
$$\text{appendright } x \ l = l \ ++ \ [x]$$
- `foldr appendright [] = ??`
- `foldr appendright [] = reverse`

So, in other words `foldr` of `appendright` to the empty list is the same as `reverse`.

It just reverses the list.

(Refer Slide Time: 07:12)

### Examples ...

- What is `foldr (++) []` ?
- Dissolves one level of brackets
- Flattens a list of lists into a single list

What would be fold the function plus, plus, the function that joins two lists together using again the empty list. So, then we will see that this actually works like concat, because dissolves one level of brackets. So, supposing have 1, 2 and 3, 4 and I take this, then it will take this plus, plus, this give me 3, 4 and it will take this plus, plus giving me 1, 2, 3, 4.

(Refer Slide Time: 07:39)

### Examples ...

- What is `foldr (++) []` ?
- Dissolves one level of brackets
  - Flattens a list of lists into a single list
- The built-in function `concat`

So, this is just the same as the built in function concat. So, we can write familiar functions in different ways using foldr.

(Refer Slide Time: 07:49)

**foldr**

$f: a \rightarrow b \rightarrow \boxed{X} b$   
 2 arg:  $\rightarrow$  result

$\text{foldr } f \ v \ [] = v$   
 $\text{foldr } (f) \ v \ (x:xs) = f \ x \ (\text{foldr } f \ v \ xs)$

- What is the type of foldr?

Diagram illustrating the foldr process:

Sequence of values:  $x_{n-1}, x_n, y_n$

Function  $f$  is applied to  $x_{n-1}$  and  $x_n$  to produce  $y_n$ .

Final output:  $y_1$

So, foldr as we saw before is defined like this. So, the question is, what is the type of this function? So, the type of this function involves first looking at the function itself. So, what does  $f$  do?, it takes two arguments and it produces a result. So, there are three possible things. It could be in general of type  $a \rightarrow b \rightarrow c$ . On the other hand, what we have seen is that the result that it produces is the right hand side of the next. So, we have, say the value  $v$ , then we have  $x_n$ ,  $x_{n-1}$ , and so on.

So, when I apply  $f$  to this and I produce a  $y_n$ , then I am going to apply  $f$  to this again. So, therefore, the output of this function must be compatible with the second argument of  $f$ . So, this must actually be another  $b$  and then at the end I am eventually going to produce  $y_1$  which is the output of the function. So, that will be a  $b$ . So, it takes a function of the form  $f$  which takes two arguments of type  $a$  and  $b$  and producing output of type  $b$ . So, I can keep repeating this process, it takes an input list, this whole input list is of type  $a$  and it produces an output of type  $b$ .



(Refer Slide Time: 09:06)

## foldr

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

- What is the type of foldr?

$$\text{foldr} :: \underbrace{(a \rightarrow b \rightarrow b)}_f \rightarrow b \rightarrow [a] \rightarrow \underline{\underline{b}}$$

So, the type of foldr is the following. So, this is my function, it takes two arguments and produces an output of the same type as the second argument. It takes an initial value which is of the same type as final output. It takes a list of inputs which are compatible with this argument of the f and produces an output which is the output of the last application. So, foldr takes a function  $a \rightarrow b \rightarrow b$  and initial value b and input list of type a and produces an output value of type b.

(Refer Slide Time: 09:37)

## foldr1

- Sometimes there is no natural value to assign to the empty list
- Finding the maximum value in the list
  - Maximum is undefined for empty list

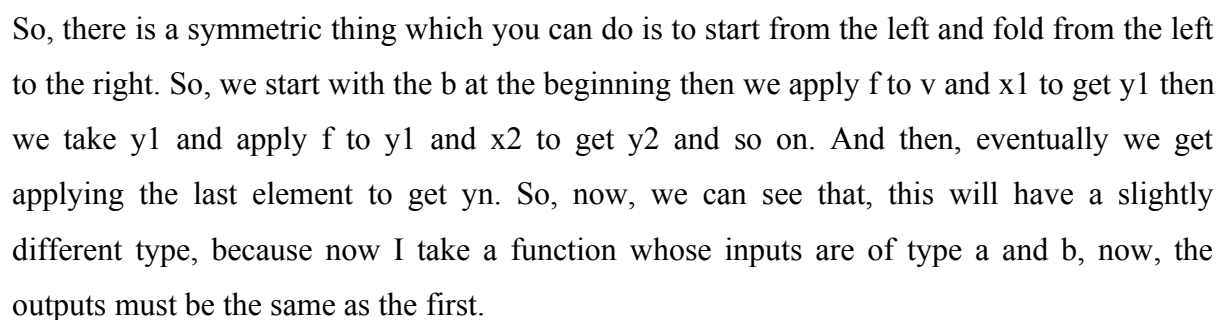
```
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

`maxlist = foldr1 max`       $\text{max} + 3 = 7$

So now in some situations, there is no natural value to assign to the base case to the empty list. So, foldr basically takes two arguments the initial value and then, the function. So, the function is sometimes may not be sensible for an empty list, for example, supposing we want

So, it makes sense to have a function which only uses a non-empty list, this in Haskell is called `foldr1`. So, `foldr1` on a non-empty list, for a singleton will just return the value itself. So, the base case for this is a non-empty list of length 1, because 0 is not allowed. It cannot have the empty list and otherwise, it behaves exactly like `foldr`. So, it takes the first argument of a non-empty list and applies `f` along with the result to bring it to the rest.

(Refer Slide Time: 11:01)



So, this must again be of type a, because again I am going to map it to b, this will again be of type a and again I am going to map it to b and finally, this is of type a type b type a. So, the

function is from  $a \rightarrow b \rightarrow a$  and then, we have a initial value of type  $a$ , we have a list of  $b$ 's and it produces an output of type  $a$  and when recursively applied, notice that, the first argument is the inductive thing of the left segment and we applied this value to what remains. So, it is slightly different from `foldr`, because we are going left to right.

(Refer Slide Time: 12:16)

### Example

- Translate a string of digits to an integer  
`strtonum "234" = 234`
- Convert a character into the corresponding digit:  

```
chartonum :: Char -> Int
chartonum c
  | ('0' <= c) && (c <= '9')
    = (ord c) - (ord '0')
```

So, here is an example of using this `foldl`, if we want to take a string of digits written using a character and represented as a number. Then it is natural to move from left to right, because as we see each digit, the number increases in length. So, our base function can be one which takes a single character and converts it to a digit. So, we have seen the function `character to number chartonum`, which checks whether `c` lies in the range 0 to 9 in characters.

Remember, we assumed that the characters are represented in tables and the digits are all contiguous in this table. So, if it lies in this range, then we just compute the number by taking its offset with respect to the table value of 0. So, 0 will be mapped to 0 , 1 will be mapped to 1 and so on.

(Refer Slide Time: 13:12)

Example ...

"234"  
 $23 \times 10 + 4$

- Process the digits left to right
- Multiply current sum by 10 and add next digit

```
nextdigit :: Int -> Char -> Int
nextdigit i c = 10*i + (chartonum c)

strtonum = foldl nextdigit 0
```

And now what we want to do is to fold this function from left to right. So, what we do is, we assume that we have say processed 2 and 3 and got 23 as a number. Now, I look at 4, so I will multiply this by 10 and add 4. So, this is how we normally do it in place value thing. So, we multiply the current sum by 10 and add the next digit. So, the next digit will be 10 times the current digit plus the conversion of the next character.

Now, if you start with the base value 0 and go from left to right then the first step is to convert the character 2 to number 2, so I will get 2. Then, the next time I will take  $2 \times 10$ , 20 and add 3, so we are get 23. So, I will get 23 then I will take 23 times 10 and add 4 and I will get 234. So, this is an example where you fold the function from the left to the right.

(Refer Slide Time: 14:12)

Summary

- Many cumulative operations on lists can be expressed in terms of folding a function through the list
- Built in functions `foldr`, `foldr1`, `foldl`

← →

So, Map and filter are functions which transform elements to elements individually, but if you want to combine the elements in the list into a single value as an output. Cumulative operation typically involves folding a function with the list, applying it from one end to the other end. So, the basic functions in Haskell which do this are foldr and foldl. So, foldr goes from right to left, foldl goes from left to right and there are versions of these functions, which do not have a sensible value for the initial empty list called foldr1.