

Introduction To Haskell Programming

Prof. S. P Suresh

Chennai Mathematical Institute

Module # 03

Lecture - 03

Lists: Map and Filter

So, we have seen that Haskell supports higher order functions, functions that take other functions as arguments and apply them to yet another argument. In the context of lists, map and filter are the two most important higher order functions that one can use.

(Refer Slide Time: 00:19)

So, to introduce map, let us look at two functions we have seen before. The first function is one that converts the entire string into upper case. So, this function is defined inductively using the base function capitalize, which converts a single letter to upper case. So, touppercase of the empty string is the empty string and if we have a non empty string, then we capitalize the first letter and then recursively upper case the list.

Another function which we have seen is one which squares all the elements of a list of integers. So, the square of an empty list is an empty list and if we have a non empty list, we square the first number and then recursively square the rest. So, in

both of these functions we are applying the given function to each element of the list. In the first case we are capitalizing every character in the string, in the second case we are squaring every number in the given list.

So, this is what the built in function map does. So, map in general takes a function f and applies it to every element of the list, so we have a list x_0 to x_k , then x_0 will be replaced by f of x_0 and so on. So, we have this kind of function which takes each element and just applies the function one element at a time. So, this produces a new list of exactly the same length as the old is, in which each x_i is replaced by f of x_i , so it maps the function to the list.

(Refer Slide Time: 01:49)

Examples

- `map (+ 3) [2,6,8] = [5,9,11]`
- `map (* 2) [2,6,8] = [4,12,16]`
- Given a list of lists, sum the lengths of inner lists

```

sumLength :: [[a]] -> Int
sumLength [] = 0
sumLength (x:xs) = length x + (sumLength xs)

```

- Can be written using map as:

```

sumLength l = sum (map length l)

```

Handwritten notes: $[[1,3],[2],[3]]$ with arrows pointing to $[2, 1, 0]$ and the calculation $2+1+0=3$.

So, here is an example, remember we saw last time that if you take an operator like $+$, then you can make it a function by using the round bracket $(+)$. And therefore, if I have plus applied to n and m , then the function with one argument consumed plus m , actually adds m to every element that is applied. So, now, if you take the function $(+ 3)$ it adds 3 to whatever it is applied to. So, if I map this onto this list, I get $(+ 3) + 2$, which is 5, $6 + 3$ is 9 and $8 + 3$ is 11.

So, this is one easy example of map, here is the similar one using multiplication. So, now, you have $2 * 2$ is 4, $6 * 2$ is 12, $8 * 2$ is 16. So, here is a slightly more involved example, supposing we have a list of lists, so say we have a list consisting of say 1, 3 and then 5 and then we get an empty list. So, what we want to do is we want to compute the sum of the lengths of these lists, so the sum of this, the length of this

list is 2, the length of this list is 1, the length of this list is 0.

So, for this we want a function that will return 3, this $2 + 1 + 0$, so it is not the length of the whole list, but the length of the inner lists. So, we have a list of any type contain, the list of lists of any type and we want to return an integer and here is the usual inductive definition. We said that the sum of the lengths of the empty list of the lists is 0, because there are no lists inside and if I have a non empty collection of lists, then I compute the length of the first one and then inductively compute the rest.

So, this is the traditional inductive or recursive definition of this function. Now, on the other hand, what we can do is, we can use maps, we say take the list and apply length to each one of them, does this map. So, this takes for example in the list that we had seen earlier, so the effect of map is to replace this by 2, this by 1 and this by 0. So, now, we have this new list, which is after we map length of l.

And then the built in function sum adds up all those things, so if I apply sum of these I will get $2 + 1 + 0$, which is 3. So, sumLength can now be successively described in terms of map and sum, rather than writing out an inductive definition like this.

(Refer Slide Time: 04:20)

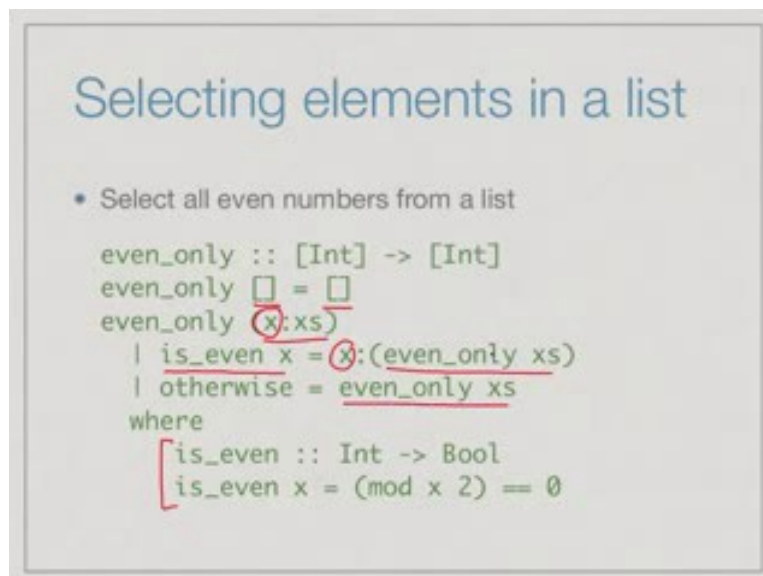
The slide is titled "The function map". It contains two bullet points. The first bullet point is "The function map" and shows the definition of the map function in Haskell: $\text{map } f [] = []$ and $\text{map } f (x:xs) = (f x) : (\text{map } f xs)$. The second bullet point is "What is the type of map?" and shows the type signature $\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$. There are blue arrows on the slide: one from the $(a \rightarrow b)$ part of the type signature to the f in the definition, and another from the $[a]$ part to the xs in the definition.

So, the function map itself can be defined inductively pretty much the way we have been doing the specific cases. So, if you map f to the empty list, then we get the empty list, if you map f to a non empty list, then we apply f to the first element of the list and recursively apply f to all the other elements. So, what is the type of map?

So, map takes of course as input some list, so this is a list of some type a.

And now this is the function that operates on elements of this list, so this function must start with an input a, but it can produce an output of any type. So, it will be in general of type $a \rightarrow b$ and the outcome of taking a list of a, and applying a function that goes from $a \rightarrow b$ is, of course produce a list of b. So, if we ask a type of map it will say, give me a function from some a to some b, give me a list which is compatible with the input type of this function. Because, I want to apply to each element of the list and it will in turn produce a list, which is of the output type of a function.

(Refer Slide Time: 05:37)



The slide is titled "Selecting elements in a list". It contains a bullet point: "Select all even numbers from a list". Below this, it shows the definition of a function `even_only` in Haskell. The function signature is `even_only :: [Int] -> [Int]`. The base case is `even_only [] = []`. The recursive case is `even_only (x:xs)`, which is defined using a conditional expression: `| is_even x = x:(even_only xs)` and `| otherwise = even_only xs`. A `where` clause defines `is_even` as `is_even :: Int -> Bool` and `is_even x = (mod x 2) == 0`. Red annotations highlight the `x` in the list pattern and the `x` in the conditional expression, and the `even_only xs` recursive call.

```
• Select all even numbers from a list

even_only :: [Int] -> [Int]
even_only [] = []
even_only (x:xs)
  | is_even x = x:(even_only xs)
  | otherwise = even_only xs
where
  [is_even :: Int -> Bool
   is_even x = (mod x 2) == 0
```

So, map takes the function from $a \rightarrow b$, it takes a list of type a and it produces a list of type b, So map is one important higher order function. Now, let us look at another higher order function, which involves selecting elements in the list. So, here is the specific example, supposing we have a list of integers and we want to select from this list of integers those that are even. So, inductive definition would say, that if I have the empty list of integers, then I get the empty list of integers.

If I have a non empty list of integers, then I have to decide whether to include the first element or not. So, I have a function here which checks whether a number is even, it just checks that the remainder of x divided by 2 is 0, so that is definition of even. So, if x is even, then include x and continue extracting the even numbers in the rest otherwise x is also excluded and just go to the rest and this is again a traditional

recursive definition of this.

But, we would like to do this in general, so here it says you want to select all the even numbers. In general we may have some other property, which is True and False of some elements and pick out all those elements for which the property is True.

(Refer Slide Time: 06:48)



The slide is titled "Filtering a list" in blue text. It contains a bulleted list and several lines of Haskell code. The bullet point states: "• filter selects all items from list l that satisfy property p". The code defines the filter function recursively: `filter p [] = []`, `filter p (x:xs)` followed by a conditional expression `| (p x) = x:(filter p xs)` and `| otherwise = filter p xs`. Below this, the type signature is given: `filter :: (a -> Bool) -> [a] -> [a]`. Finally, an example is shown: `even_only l = filter is_even l`, where `is_even` is circled in blue. A handwritten note in blue ink below `is_even` reads `:: Int -> Bool`.

```
Filtering a list
```

- filter selects all items from list l that satisfy property p

```
filter p [] = []
filter p (x:xs)
  | (p x)      = x:(filter p xs)
  | otherwise = filter p xs

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

even_only l = filter is_even l
                :: Int -> Bool
```

So, this is called filtering, so in filtering we take a property p, so property p is a function that takes a type and maps it to a Boolean value. So, it takes say integers and decides whether they are even or not, take letters, characters and decide whether they are vowels or not for example. So, filter takes a property and a list and it extracts exactly those items in the list that satisfy property p. So, filter takes p and if I have to filter an empty list [] with the property p, I get nothing, because no elements are there to satisfy the property.

If I have a non empty list, then I check the property on the first element, if it is True I include the first element and continue, otherwise I exclude. So, in our previous case the property was `is_even`, but this is a generalization. So, as we said a property is just something, which takes the type of the underlying list and decides whether or not each element satisfies the property.

So, it takes a function which maps the type a to Bool, it takes an input list and produces an output list, which is the subset or sub list of the input list that of the same type. It is not like map, which produces a new list, which is the result of transforming the elements by a function. So, in a map each element is transformed

by a function from $a \rightarrow b$, so you get a list of these, in filter you are not transforming anything, you are really selecting a sub list, so the output list and the input list have the same type.

Of course, it is possible that none of the elements in the list satisfy p , in which case the output list maybe empty, where it will be of the same type. So, if we go back to our function `even_only`, then the property in this case is the function `is_even`, if you wrote before. So, `is_even` remember it takes integers and produces Booleans, just checks whether the remainder is 0 when divided by 2. So, if I filter a list given this function, then I will extract exactly the even numbers from the list, so this is a much more succinct way of writing the same function.

(Refer Slide Time: 08:59)

The slide is titled "Combining map and filter". It contains two bullet points and two code snippets. The first bullet point says "Extract all the vowels in the input and capitalize them". The second bullet point says "filter extracts the vowels, map capitalizes them". Below these are two code snippets. The first snippet defines `cap_vow` as a function from `[Char]` to `[Char]`, and its implementation is `cap_vow l = map touppercase (filter is_vowel l)`. The second snippet defines `is_vowel` as a function from `Char` to `Char`, and its implementation is `is_vowel c = (c=='a') || (c=='e') || (c=='i') || (c=='o') || (c=='u')`. Red annotations highlight the `filter` and `map` functions in the code, and a red bracket groups the `is_vowel` definition.

Combining map and filter

- Extract all the vowels in the input and capitalize them
- filter extracts the vowels, map capitalizes them

```
cap_vow :: [Char] -> [Char]
cap_vow l = map touppercase (filter is_vowel l)

is_vowel :: Char -> Char
is_vowel c = (c=='a') || (c=='e') ||
              (c=='i') || (c=='o') ||
              (c=='u')
```

So, very often map and filter occur in conjunctions, so very often what we want to do is to take a given list, extract some elements from that list, which satisfy given property and then transform those elements to some new elements. So, here is an example supposing we want to extract all the vowels. So, this is the filter and then we want to capitalize these letters, so that is a map. So, filter extracts the vowels and map capitalizes them, so we have first a filter function called `is_vowel` similar to `is_even`, which just checks whether the character is an a e i o or u.

So, we first apply filter to the list with `is_vowel` as the property, this will result in a list, which has exactly those characters, which are vowels in the original list. And then, we use our earlier function `touppercase` which capitalizes every element in a

string to apply to this list, which contains the vowels. So, filter followed by map is a very common form that we will find in many functions that we use.

(Refer Slide Time: 10:06)

Combining `map` and `filter`

`[1, 2, 6, 9, 11, 14]`
✗ — — ✗ ✗ —
↓
4 36 196

- Squares of even numbers in a list

```
sqr_even :: [Int] -> [Int]
sqr_even l = map sqr (filter is_even l)
```

So here is another simple example of filter and maps. Supposing we want to square all the even numbers. So, we first filter the list by getting only the even numbers out of them, then we map the square function to this list. So, this will square each individual element of that list and give us the squares of all the even numbers, so if we have list [1, 2, 6, 9, 11, 14], then first filter will give us 2, 6 and 14, others are excluded. So, I get 2, 6 and 14 and then map will give me 4, 36 and 196.

(Refer Slide Time: 10:49)

Summary

- `map` and `filter` are higher order functions on lists
- `map` applies a function to each element
- `filter` extracts elements that match a property
- `map` and `filter` are often combined to transform lists

So to summarize, map and filter are extremely useful higher order functions of list. So, maps take a function and a list and applies the function to each element of the list. Whereas, filter takes a property that evaluates to True or False, and extracts elements from the list that match the property, that is those elements for which the property is True and we often use these in combination, so to extract the sub list and then apply a function to that list.