Introduction To Haskell Programming

Prof. S. P. Suresh
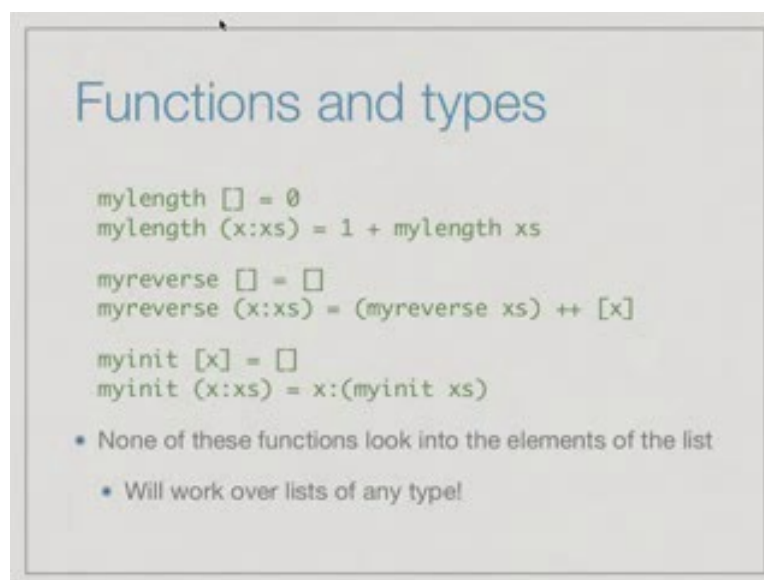
Chennai Mathematical Institute

Module # 03

Lecture – 02

Polymorphism and Higher Order Functions

Let us take a closer look at types in Haskell.

(Refer Slide Time: 00:06)

Early on we wrote some functions on list, which we defined as functions over list of integers. For instance, we wrote this function length, which inductively assigns 0 to the length of the empty list and adds 1 for each element in the list. We also wrote this similar inductive function to reverse a list. And finally, we wrote the reverse of the head function, which takes the last element; I mean the tail function, which takes the first segment of the list, leaving off the last element.

Now, all of these functions have one thing in common, which is that they do not actually bother about the values in the list, so it is really a structural property of the list. The length of the list really just depends on how many elements there are, when you reverse the list, we just take the elements in the list and reverse the order. Similarly, when you take the initial segment of a list, you just drop the last element.

So, in all these things, you can actually assume that similar things will work regardless of the type of element in the list. So, if you think of a list as a list of boxes in some sense and then if you need a value in the list, you need to open the box, so none of these functions actually need to open the boxes, they can just move the boxes around or select a few of the boxes or one of the boxes and so on.

(Refer Slide Time: 01:26)
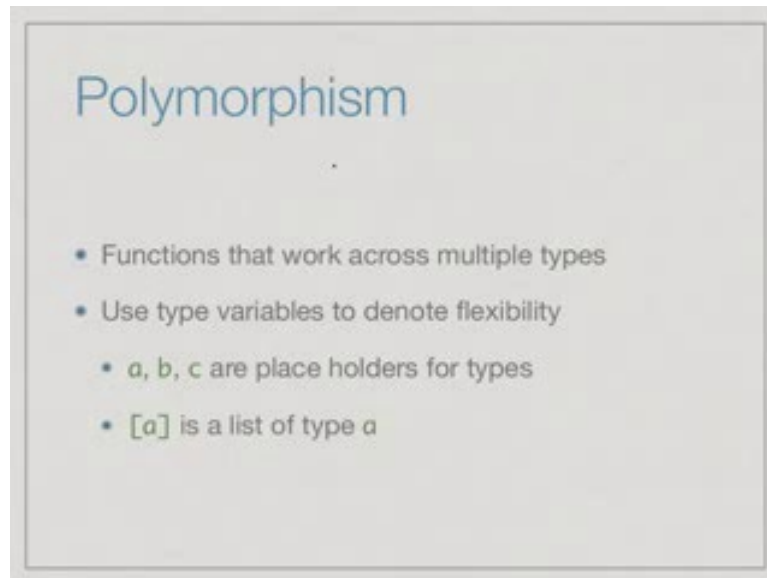
So, it would be nice if we could assign within the type system that Haskell uses for each function. The type which will allow a function to work across multiple input types like this. So, we would like reverse for instance to work on list of Int or list of Float or even in list of list. So, as long as the underlying value is of a uniform type, so in Haskell, we use type variables, usually a, b, c; letters like these as place holders for type. So, if we say that a function takes as input type a, it means a can stand for any input. And if a stands for any input, then if you put square brackets around a, [a], then we denote a list of that type.

So, if you go back now to our list functions, what does my length do, mylength takes any list and produces the length, the length is a number, so the output type is Int, the input type is unconstrained, it can be a list of any type of a. So, we write the type as [a] -> Int, reverse on the other hand, takes a list and produces another list. So, once again it takes an arbitrary list, but it produces a list of the same type.

So, it is important that the a on the left and the a on the right are actually the same a, it means that, if I give it an input of [Int], it will produce an output which is a [Int]. If I give it a list of float, it will produce a list of float as output, you cannot change the type, but the type itself is not fixed in advance. And finally, if you look at myinit, so what it does is it actually produces, so this is a small mistake here.

So, it takes again a list of a, [a] and produces the list of a, [a] and if you wrote a function like head for instance, then it would take a list of a and produce a single value of type a. So, the important thing to note is that, if I have a variable, then wherever it appears in the type it must be instantiated by the same type.

So, let us digress a little bit about notation of operators and functions in Haskell. So, we have seen arithmetic operators like +, -, /, we also have you know Boolean operators like OR and AND, so we could have you know x and y and so on. So, these are operations which operate on two values, but we write them in this infix notation; that is we put the operation in between the arguments. So, we have two arguments and the operator comes in between, so the operator is said to be infix.
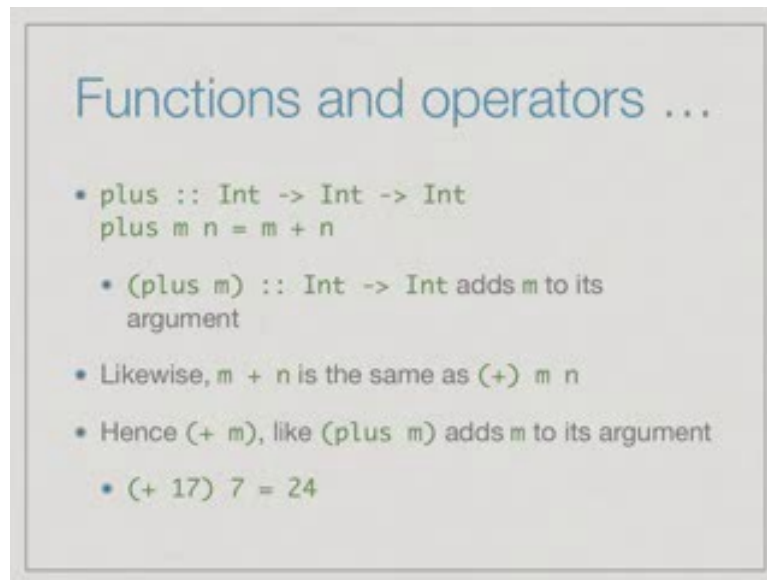
On the other hand, we have seen functions, which are predefined like div and mod, which are written as functions. In other words, you write the name of the function and then, you write the arguments. So, this is what is called prefix notation, because the function comes before the arguments. Now, sometimes you may want to switch between these two notations or two views.

So, in particular, we can take any binary operator which we have defined and make it into a function just by putting round brackets around it. So, the operator '+' can be written as a function by writing (+) . So, this expression (+) 3 , 5 is exactly the same as 3 + 5 and so on. So, in this way, we can use this notation to convert operators to functions and we can also do the reverse, provided of course, the function takes two arguments, because otherwise it is not clear, what it means to be infix, infix means it is between two arguments.

So if we have binary functions like div or mod, then we can use this reverse quote ` , this is the back quote key, this is not the normal apostrophe but the back quote, which we will find

on your keyboard. So, if we enclose the name of the function within these reverse quotes, then we can use them as operators. So, if you want to say 11 mod 5 and not mod 11, 5; then we write   11 `mod 5, so here we have written 7 `div 5, 11 `mod 3 and so on.

(Refer Slide Time: 05:40)

So, this is useful in the sense that we have seen, suppose we have written a function plus, which takes two integers and adds them. What we saw is that because of currying plus actually consumes the arguments one at a time. So, in an intermediate stage, when it has consumed the first argument, we have a new function which we can write call plus m n, which actually takes, whatever it takes as a second argument, namely the n and it will add m to it.

So, m is now fixed, so in the same way, we can think of this using the normal plus operator. So, m + n is the same as this (+) m n. So, this (+) is the function which exactly like the p l u s, plus, we wrote above, which takes two arguments and if you take one of them, then it becomes the function plus m. So, (+) m adds m to its arguments.

So, you can try this out in the Haskell interpreter in ghci. So, if you take something like (+ 17) and you applied it to 7, written as (+ 17) 7 , then (+17) will add 7 to 17 and give you 24. So, using the fact that we can convert operators to functions by using the round bracket, we can also get partially instantiated functions, where one of the arguments has been fixed and use these, when we manipulate functions in Haskell.

Now, manipulating functions in Haskell takes us to this notion of a higher order function. So, in Haskell, there is no particular problem or specific thing that we need to do to pass a function as an argument to another function. So, here is the most obvious function, we could write like this apply f x = f x . So, we can take a function called apply, which takes two arguments, and applies the first one to second one.

So, it really reads the function as its first argument, it reads the value as the second argument and applies the function it gets to the value it gets. So, it just says given f and given x return the value f of x. So, here for instance, if we say something like apply plus 17 7; then this should reduce by the application of plus 17 to 7 to the value 24, this is the intention.

So, what is the type of this function, well the first argument is the function and we wanted to be possible to send all possible functions to apply. So, if we make no assumption about the function, then it takes an input type and it takes an output type, which may be different. So, generic function f has type from a -> b, where a and b are possibly different types.

So, there is no constraint a and b may be the same type, may be different types, we do not constrain it in any way. Now, given that f takes an input of type a and produces an output of type b, this constrains the type of x, because x is now going to be passed to f. So, f takes x, so x must be an a and the result must be a b. So, the argument x and the output must be compatible with this choice.

So, this must be of type a and this must be of type b. So, this gives us the following type, so at this function, so this is not f but apply. So, apply takes as its first argument, some unknown function f and it takes as a second argument, a value to be passed to this function. And it produces an output, which is the application of x, f to x, which because f has type a to b, this is of type b. So, apply has type from a function a to b and an input of type a, produce an output of type b.

(Refer Slide Time: 09:33)

So, higher order functions will turn out to be very useful. So, here is a very familiar thing that you would have seen before. So, suppose we want to sort a list of objects, and the heart every sorting algorithm is a comparison between objects. So, we need to take pairs of objects and order them in the right way. So, in some situations, it might be important to know how to compare these objects, if we have pairs, then we use only one component, we use dictionary ordering and so on.

And also we might want to know in which direction to order, we want to order them in an ascending order or descending order and so on. So, to be fully flexible, what we would like is that, we have the sort function and it should take actually two inputs, it should take the compare function and it should take the list and then, use this compare function to sort this list, according to the sorting algorithm.

So, by changing the compare function for instance, we can make it go from ascending to descending and so on. So, this is a natural situation in which we pass a function to another function and we will see other examples of this.

(Refer Slide Time: 10:43)
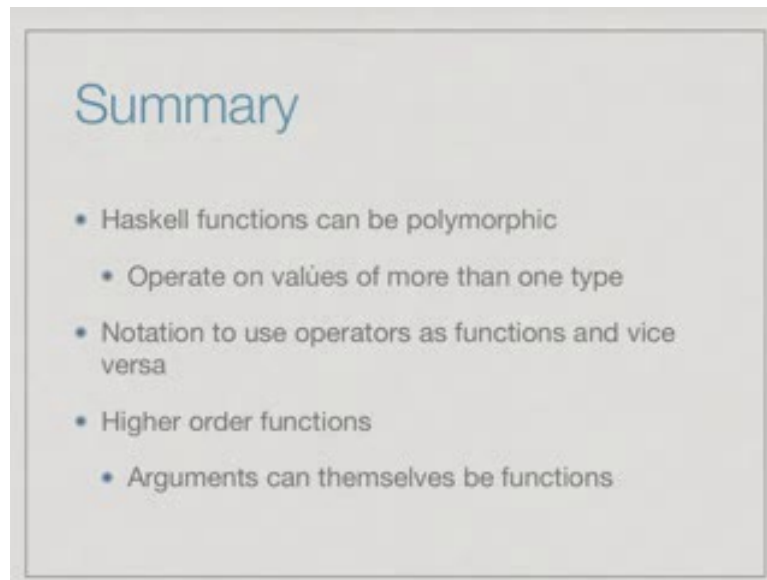
So, to summarize Haskell functions typically those we have seen, which are structural functions on list, things like take the length or reverse it or take an initial segment or a final segment, can be polymorphic. In other words, they can operate on list of more than one type. We have also seen that we can use this round bracket and the backquote to convert operators to functions and vice versa.

And finally, we have seen that in Haskell, it is no big deal to pass a function to another function. So, we have higher order functions. So, to speak for free and this is very useful and we saw an example in the context of sorting and we will see more examples in the lectures to come.