Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

Module # 01

Lecture – 05

Currying

Let us now turn to the mysterious notation we have been using for functions with multiple inputs.

(Refer Slide Time: 00:08)

So, what we have been seeing is that when we have a function which takes two inputs like plus. Instead of using the familiar notation, where we use a bracket and we put the arguments inside the brackets separated by commas, we just write the arguments one after the other separated by spaces. Now, when we write a function like plus in the usual way, we indicate that it takes two arguments and this is called its arity. So, the arity of a function is how many arguments it takes.

So, in this case we say that plus is a binary function. So you have binary functions that take two arguments, unary functions take only one argument and so on. So, this becomes part of the definition of the function and when we use the function we have to ensure that we supply the right number of arguments. So, this is the conventional point of view. Now our radical departure from this is to assume let all functions take only one input and this is the principle behind the notation that we have been using and we will see in a minute how it works and

just as a piece of terminology, this style of writing functions where every function is the function of only one argument is called currying.

So, it has nothing to do with cooking, but rather it is named after the logician Haskell Curry, who made it popular. So, Haskell Curry is not in fact, the person who invented it. The person who invented it was another logician called Schonfinkel, but Haskell Curry was a logician who made this style of notation for functions popular. And another piece of interest for us is that the language Haskell, you might have wondered where the name Haskell came from, where Haskell is actually named after the logician Haskell Curry.

(Refer Slide Time: 01:53)

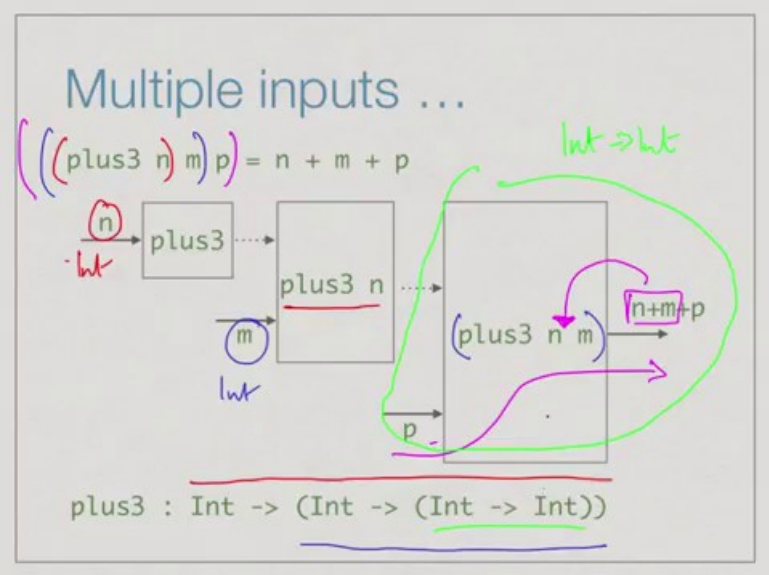
So, let us look at these two view points of functions. So, on the left we have a familiar plus which takes two arguments n and m and produces an output or the answer n+m. The right is a picture of how we are supposed to think about the curried version. So, what we are saying now is that plus as we have defined it takes only one argument. So, the one argument it takes is the first argument, so effectively we start by consuming this n. So, plus consumes n, so that is this box and it produce the new function in which n has now been observed.

So, now, we have a function which will add n to whatever it gets, so that is the principle. So, instead of consuming multiple arguments in one shot, you consume one argument at a time and then you transform yourself into a new function in which part of the functionality is internalised. So, plus has consumed the first argument and became a fixed function called plus n, which will add n to whatever argument it gets. So if the first argument to plus was 2, then now this new function we call plus 2. It will add 2 to whatever it gets.

And now this new function is going to consume the m. This is now going to take this m, add n to it, which has already been built in, and give us n+m. So, the idea in currying is that instead of multiple arguments you get a sequence of functions. So, you consume one argument at a time, each argument transforms the function in some way by internalizing the most recent argument and creating a new function which will consume one more argument and so on.

So, let us try to look at the type of the new plus that we have defined. So, if we start from the end the last thing that happens is function plus n which takes an Int as an input and produce an Int as output. So, this particular function here is Int to -> and this whole thing is the output of plus. So, therefore, the type of plus is something that consumes an Int and produces this function. So, that is why we get that the type of plus is from an input Int to an output Int -> Int, that is, Int->Int->Int. So, this output Int -> Int is this box and the input Int is the original Int. So, working backwards we can take a curried function and work from the last box backwards to construct its type like this.

(Refer Slide Time: 04:34)



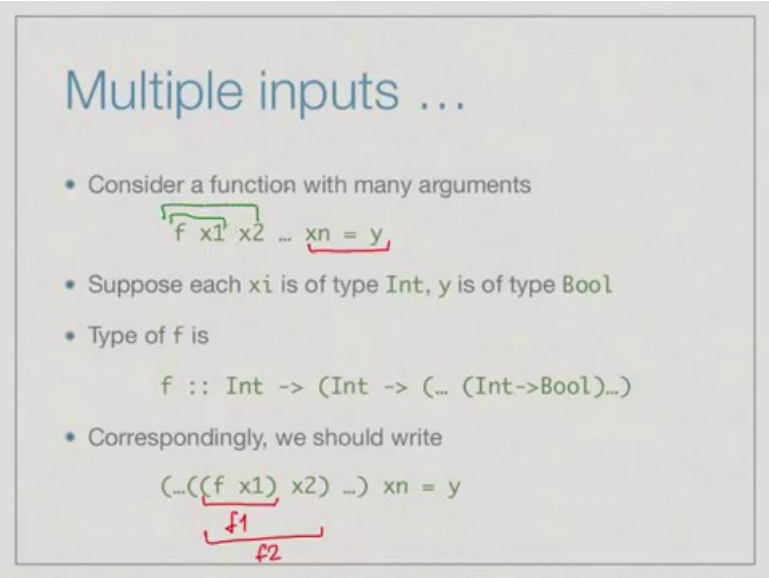
Let us look at another example one that takes maybe 3 inputs. So, supposing we have a plus3 which will add 3 numbers. So, now, again in the same way when I first consume the first n I will get something which internalizes the n, so now it will add n to whatever it is going to get as a next input, now we are not done. So, we will now consume one more input which is this one and we will now have something which adds n + m to whatever it gets.

And finally, when we consume the third input here we will have this function which takes us

from a number p to the number n + m + p and the point is this n + m is kind of built into the function. It has already been hard wired in a sense, because we have consumed n and m as previous inputs. So, now, again working backwards, so this last thing is the last box here, so this is an Int -> Int function, so that is this type.

So, therefore, if we go to the previous box, so this now takes as input an Int and produces that. So, that is this function and finally, the outermost takes an Int and produces Int -> Int -> Int and therefore, the type of plus3 this is whole expression. So, if we have a curried function we consume the inputs one at a time, each input transforms a function into one, where one value is frozen in some sense. So, the function now becomes one where one argument is fixed. So, we keep consuming one argument at a time and keep transforming the function. So that, it now has some values fixed in to it and we can recover the type of the function by working backwards from the last box to the first box.

(Refer Slide Time: 06:20)



So, in general we could have a function which takes say n arguments and produces an answer. So, suppose in this particular function that we are looking at each of these inputs 1 to n is Int and say the last one is of type Bool then by our earlier description we would start by working backwards. So the last box would take the last input xn and produce y. So, this would be Int->Bool, so this is this last thing and then the previous box would have taken xn-1 and produces function that would be Int -> Int -> Bool and so on. So, we will get this nested thing sequence of Int-> Int-> Int-> Int and finally with an Int -> Bool.

And logically if we look at the expression, the original function f first consumes x1 and it

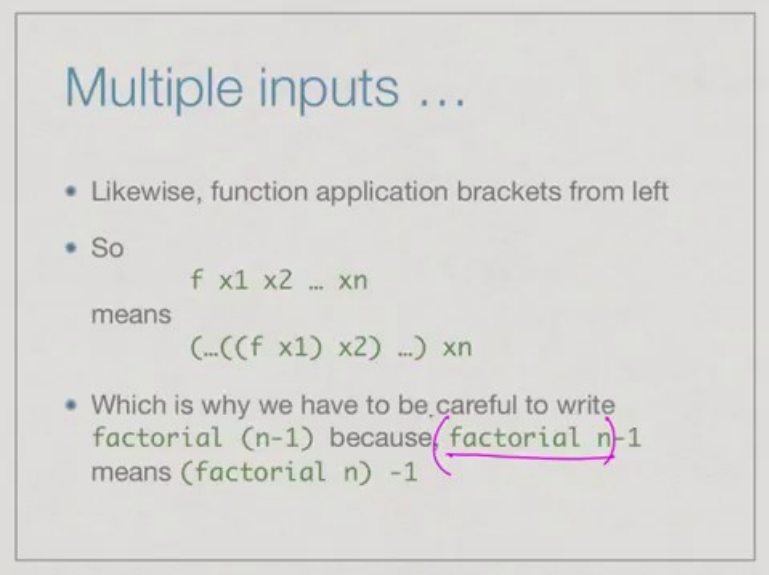
becomes a new function. So, this is our second box, so this is like an f' or an f1. f1 consumes x2 and this becomes a new function which has x1 and x2 into it, we call f2. So, we have this implicit bracketing of the types in one sense, where the innermost bracket corresponds to the right most function, the last input in the last output and we have the corresponding bracketing of the way the function is used which is that we first consume x1 then we consume x2 and so on.

(Refer Slide Time: 07:50)

Now, fortunately Haskell knows this, so there is an implicit bracketing which Haskell uses depending on the type of expression that we used, we are familiar with the implicit bracketing for arithmetic for example, we have this BODMAS. So, we have in BODMAS it says that if I write for instance 7 + 2 * 3 then it is not going to be 9 * 3, this is not a correct answer, so this is not 9 * 3 rather it is 7 + 6. So, there is a precedence which says that it is bracketed like this. So, that says division and multiplication are bound tighter than addition and subtraction.

Now, in a similar way when it sees expression like these arrows, then Haskell knows that it must bracket them in a particular way and in particular it will bracket this starting from the right. So, it will first put bracket only to Bool then around the previous one and so on. So, it will produce from the upper expression without any brackets a lower expression and we can freely use the upper expression without worrying about all these messy nested brackets.

(Refer Slide Time: 08:50)

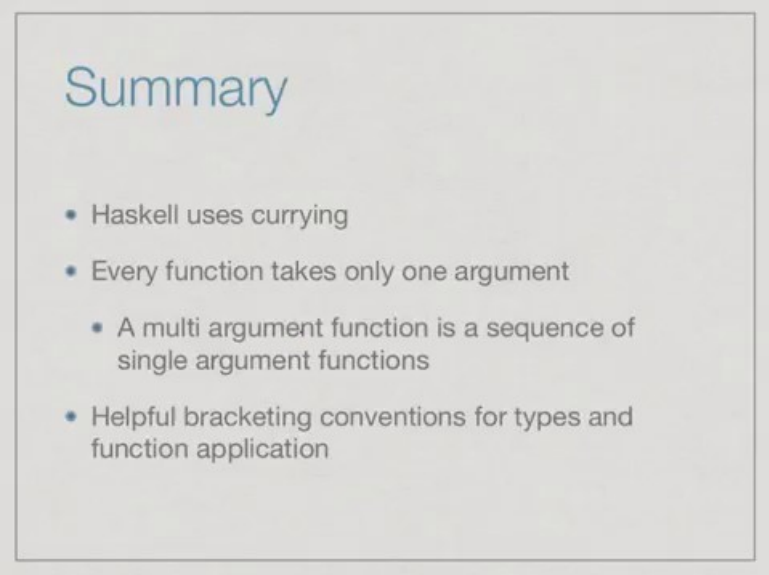


The same way when we write a function with arguments as a call to the function, it will implicitly assume that the function is bracketed from the left. So, it will put in these brackets it will start with bracket around x1 then a bracket around x2 and so on and finally, it will produce bracket around this. So, we do not have to worry about this bracket, so that is very nice for us, so it means that the built in bracketing rules in Haskell take care of the usual thing that we expect. So, unless we want to do something unusual, we do not need brackets.

Now, we have seen an example where the scan creates a problem, remember when we defined factorial we had to be careful to put a bracket around the n-1 and that is precisely because of this rule. Because, Haskell when it sees factorial n - 1 without bracket will first take factorial and bind it to nearest thing and this is like having two different operators, one is factorial with an argument and the other is subtraction.

So just like division and multiplication will get bound ahead of subtraction, so will the function call. So, this becomes (factorial n) - 1 and this is not what we expect, so we have to be a little aware of how this bracketing is done. Because, in such situations we may need to insert brackets to ensure that Haskell is, understanding the expression the way we intend it.

(Refer Slide Time: 10:07)



So, to summarize Haskell uses currying as a notation for functions and in currying, the basic simplification is that functions do not have arities, we do not have to say a function is a binary function or a k-ary function taking k arguments, every function takes only one argument. So, if a function takes multiple arguments it consumes these one after the other, each argument transforms the function by internalizing that argument and making it into a new function where one of the arguments is frozen.

So, this becomes very convenient we will also see that we can use these intermediate functions in other contexts. So, actually if we define a function of two arguments then a function in which only one argument is provided in currying is partially another function, let us say if I take plus m n and I feed 7 then plus 7 is a new function which will add 7 to whatever it gets. I can actually treat this partially instantiated function as a real function in many contexts we will see.

And associated with currying is the implicit bracketing which is right to left for types and left to right for function application, but fortunately this bracketing is built into Haskell's bracketing rules along with arithmetic and Boolean expressions. So, we do not have to use brackets unless we really want to disambiguate something.