

Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

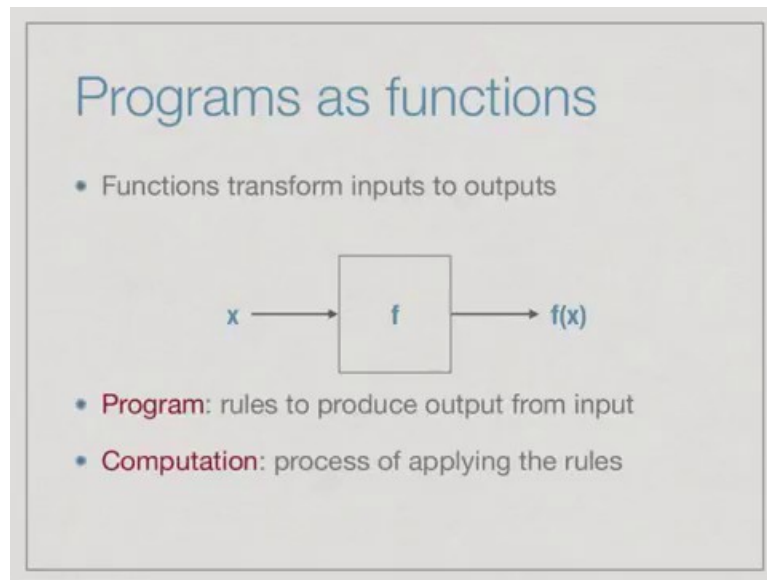
Module # 01

Lecture – 01

Functions

So, welcome to the first lecture of the course Functional Programming in Haskell.

(Refer Slide Time: 00:06)



So, functional programming starts with the point of view that the program is a function; abstractly a function can be thought of as a black box; that takes inputs and produces outputs. In other words, a program is a function that transforms inputs to outputs. So, when we write a program in a functional programming language, what we are doing is specifying the rules that describe how to generate a given output from a given input. And when we compute in a functional programming language, we apply these rules to the input that is given to us in order to actually produce the output that is expected.

(Refer Slide Time: 00:50)

Building up programs

How do we describe the rules?

- Start with built in functions
- Use these to build more complex functions

So, the first thing we have to ask ourselves is, how do we build up these programs. Now, it is impossible to start from nothing, so we have to assume that we have some built in functions and values. So, we assume that we begin with something that is given to us and then, we use these built in functions and values to build more complex functions.

(Refer Slide Time: 01:17)

Building up programs ...

Suppose

- ... we have the whole numbers, $\{0, 1, 2, \dots\}$
- ... and the successor function, `succ`

`succ 0 = 1`

`succ 1 = 2`

`succ 2 = 3`

...

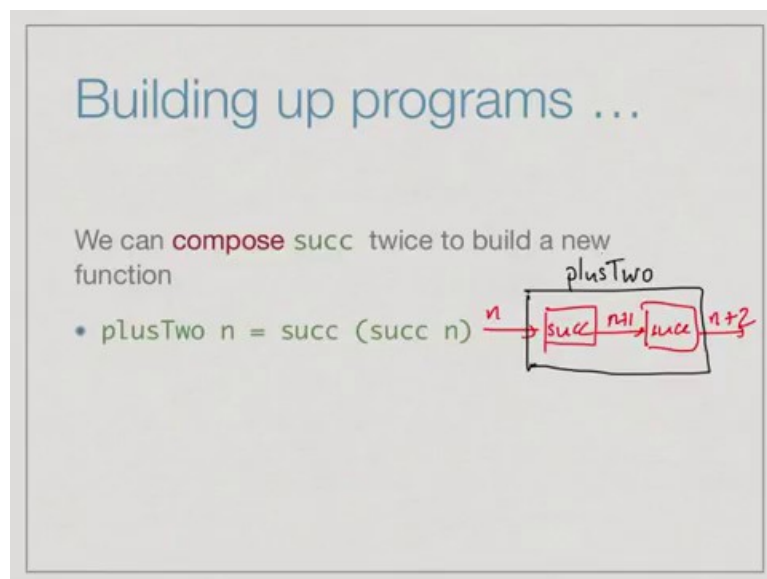
- Note: we that write `succ 0`, not `succ(0)`

$f(x)$
 $f \cdot x$

So, let us look at a concrete example. Suppose we start with the whole numbers, that is the integers 0, 1, 2 and so on, the non-negative integers and suppose the only thing that we know how to do with these numbers is to add 1. So, we have a successor function, which you can think of as plus 1 function. So, it says that 0 plus 1 is 1. The successor of 1 is 2, so 1 plus 1 is 2 and so on.

So, this is what is given to us. We have the whole numbers and the successor function. Note by the way, that we are using a slightly non-standard notation for the functions. Normally one would write successor of 0 with brackets like this $\text{succ}(0)$. So, we write $f(x)$ normally, but in functional programming, we will see that, we will normally write $f\ x$ eliminating the brackets. So, this has two advantages, one is you have fewer brackets to worry about, but it also has some interesting technical advantages, which we will describe in a later lecture in this week.

(Refer Slide Time: 02:18)



So, now, that we have a successor function, we can apply twice to an input for example, and I get a new function which adds 2. So, $\text{plusTwo } n$ takes n , apply successor and then applies successor to that, so it is as though we had two boxes with us called successor. So, we feed n here, then we get $n + 1$, we feed it to the second box and we get $n+2$ and what we are saying is that, now we can take these two boxes and call this outer box plusTwo . So, this is what it means to compose two functions, you take the output of the first function and feed it to the second function. So, in this case, we have composed the same function twice, we have taken the successor, of successor.

(Refer Slide Time: 03:05)

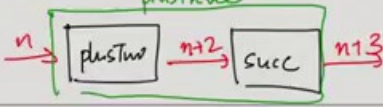
Building up programs ...

We can **compose** **succ** twice to build a new function

- **plusTwo** $n = \text{succ} (\text{succ } n)$

If we compose **plusTwo** and **succ** we get

- **plusThree** $n = \text{succ} (\text{plusTwo } n)$



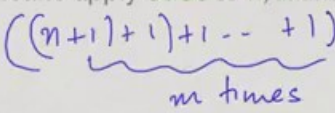
However, we can also combine two different functions, for instance we can take the **plusTwo** which we have defined and feed its output to successor. So, we have **plusTwo** and now, we have successor. So, we already know that, if we take a number n and feed it to **plusTwo**, we get $n + 2$ and then, we feed it to successor and we get $n + 3$ and now, this gives as a new box, which we called **plusThree**. So, in this way we can combine functions by function composition, which is well known to us from mathematics.

(Refer Slide Time: 03:42)

Building up programs ...

How do we define plus?

- **plus** $n \ m$ means apply **succ** to n , m times



But, now suppose we want to extend our definition of **plusTwo** and **plusThree** to **plusFour**, **plusFive**, **plusSix**. In general, we want to define **plus** with two arguments n and m , where we

mean that when we say plus $n\ m$; we apply successor to n , m times. In other words, we start with n and then, we do plus 1 and then, we do plus 1 and then, we do plus 1. So, totally we add 1 to n up till m times.

So, this is what plus means and this is how we were taught the definition of plus when we were in kindergarten, you just keep adding 1 by 1 by 1 until m times. So, how would we describe this in our setting, because we need to describe this family of plus 1's.

(Refer Slide Time: 04:29)

Building up programs ...

How do we define plus?

- $\text{plus } n\ m$ means apply succ to n , m times
 - Again note: $\text{plus } n\ m$, not $\text{plus}(n,m)$
- $\text{plus } n\ 1 = \text{succ } n$
 $\text{plus } n\ 2 = \text{succ } (\text{plus } n\ 1) = \text{succ } (\text{succ } n)$
 ...
 $\text{plus } n\ i = \underbrace{\text{succ}(\text{succ}(\dots(\text{succ } n)\dots))}_{i \text{ times}}$
- How do we capture this rule for all n, i

So, by the way again note this notation, we don't write a function of two arguments with bracket as usual, but we just write the arguments one after the other. So, plus followed by the first argument followed by the second argument, you can just think of it right now as a peculiar syntax which eliminates arguments, brackets and commas. But, as we will see this is a very interesting way of thinking about functions from a computational stand.

So, what are the rules for plus, well, we know that plus $n\ 1$ is same as successor, the built in function add 1. Plus $n\ 2$ is, we have seen the successor of successor, but we can think of $\text{succ } n$ as plus $n\ 1$. So, we are taking plus $n\ 1$ and then, applying successor to that. So, in the same way, plus $n\ i$, as we saw before will apply the successor i times and the question is, how do we write a rule which captures this mysterious dot, dot, dot which says do something a fixed number of times, but that fixed number of times depends on the value of the argument.

(Refer Slide Time: 05:33)

Inductive/recursive definitions

- $\text{plus } n \ 0 = n$, for every n
- $\text{plus } n \ 1 = \text{succ } \underline{n} = \text{succ } (\text{plus } n \ 0)$
- Assume we know how to compute $\text{plus } n \ m$
- Then, $\text{plus } n \ (\text{succ } m)$ is $\text{succ } (\text{plus } n \ m)$

$$n + \underbrace{(m+1)}_{(m+1)} = \underbrace{(n+m)}_{n+m} + \underline{\underline{1}}_{\text{given}}$$

So, this brings us to the realm of inductive or recursive definitions. So, an inductive definition is one where we specify a base case and then, we specify the value for larger arguments in terms of smaller arguments. So, for instance, we know that, if we add 0 then we do nothing, so $\text{plus } n \ 0$ is always n . So, this is the base case, we do not have to do any computation, we just return the first argument.

Now, $\text{plus } n \ 1$ consist of adding 1 to n , but we can also think of applying the previous value 0, so we get $\text{plus } n \ 0$. So, we just do not think of it as n , but we think of it as the base case and now, we apply successor to the base case. Similarly, in general, if I want to add something to $m + 1$, then assuming that I know how to do $n + m$, then I can add 1 to that. So, this is just saying that the value of $n + m + 1$ is the same as the value of $n + m$, which I know how to do inductively, plus 1 which is given to me. So, this is my given function.

So, this is the basis of inductive or recursive definitions. That is, you take the base case, whose value is obvious and for larger values, you describe it in terms of operations that you know how to do plus the operation you are trying to define on smaller values, which is inductively known to be true.

(Refer Slide Time: 07:07)

Computation

0 1 2 ...

$\text{succ } 0 = 1$
 $\text{succ } 1 = 2$

$\text{plus } n (\text{succ } m) = \text{succ } (\text{plus } n m)$

- Unravel the definition
- $\text{plus } 7 \ 3$
 - $= \text{plus } 7 (\text{succ } 2)$
 - $= \text{succ } (\text{plus } 7 \ 2)$
 - $= \text{succ } (\text{plus } 7 (\text{succ } 1))$
 - $= \text{succ } (\text{succ } (\text{plus } 7 \ 1))$
 - $= \text{succ } (\text{succ } (\text{succ } (\text{plus } 7 \ 0)))$
 - $= \text{succ } (\text{succ } (\text{succ } (\text{succ } 7)))$
 - $= \text{succ } (\text{succ } 8)$
 - $= 10$

base case $\Rightarrow 7$

So, how does computation work, well it just unravels the definition. So, supposing I want to add 7 to 3, now in our universe, we have 0, 1, 2, etc and we know that $\text{succ } 0$ is 1; $\text{succ } 1$ is 2 and so on. So, I can think of 3 not as 3, but as $\text{succ } 2$ and now, I have a rule which says, $\text{plus } n (\text{succ } m)$ is equal to $\text{succ } (\text{plus } n m)$; So, if I have $\text{plus } 7 (\text{succ } 2)$, this is $\text{succ } (\text{plus } 7 \ 2)$; when I plug in 7 for n and m for 2.

Now , once again I can expand this : 2 as $\text{succ } 1$ and apply that rule again , and I get $\text{succ } (\text{succ } (\text{plus } 7 \ 1))$ and then, once again I can replace the number 1 by the expression $\text{succ } 0$ and then, I get $\text{succ } (\text{succ } (\text{succ } (\text{plus } 7 \ 0)))$. Now, this is the base case which is 7. So, I get $\text{succ } (\text{succ } 7)$ and then, if I continue with this computation, this will give me 8 by the built in rule, this will give me 9 by the built in rule and this will give me 10 by the built in rule.

So, this is how I will get $\text{plus } 7 \ 3$ equal to 10. The important thing to note in this is that computing the value 10 does not involve understanding anything about numbers, it is just syntactically replacing expressions by expressions and this is something that we will see more formally later. So, this is how computation works in a functional programming language, you have rules which tell you how you can replace one expression by other expression and you keep replacing expressions, until you reach the value which cannot be simplified.

(Refer Slide Time: 09:06)

Recursive definitions ...

Multiplication is repeated addition

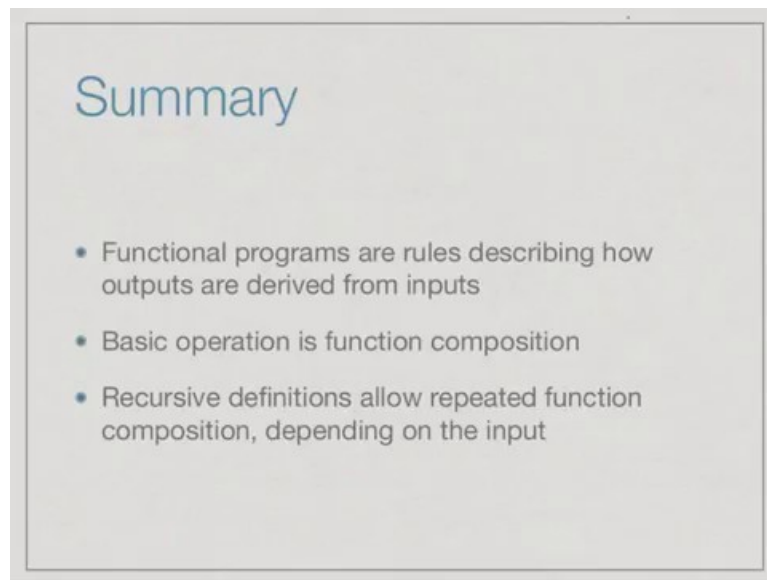
- $\text{mult } n \ m$ means apply $\text{plus } n, m$ times
- $\text{mult } n \ 0 = 0$ for every n
- $\text{mult } n \ (\text{succ } m) = \text{plus } n \ (\text{mult } n \ m)$

$$n \cdot (m+1) = \underbrace{n}_{\text{known}} + \underbrace{(n \cdot m)}_{\text{Inductive}}$$

So, let us look at another recursively defined function. So, just as addition is repeated application of the successor function, multiplication as you may remember from school is repeated addition. When I say multiply m by n , it means add n to itself m times. So, we have to apply plus n , m times starting from 0. So, the base case says that if I multiply any number by 0, then I will get 0.

So, multiplying n by 0 is 0 for every n and now, $n * (m+1)$ is just $n + (n * m)$. So, this is the inductive case, I know how to do this, because this is smaller and this is now a known function, because we have already defined plus. So, plus was not a given function, successor was the given function, but we already defined plus in terms of successor. So, now, we can use plus to define multiplication.

(Refer Slide Time: 10:13)



So, to summarize, a functional program describes rules to tell us how to compute outputs from inputs, what we have seen is that the basic operation that we use in functional programming is to combine functions. So, we use function composition where we feed the output of one function as the input of another function.

Often we have to apply this function composition more than once, but not a number of times that we know in advance. So, when we did `plusTwo` and `plusThree`, we knew exactly how many times, we have to compose the function, but when we wrote the general `plus` or the general `multiply`, we have to apply these functions depending on the value of the argument. So, for such functions, we saw that recursive definitions are a good way to capture the dependence of the number of times the function has to be composed based on the input value.