

Introduction To Haskell Programming

Prof. S. P. Suresh

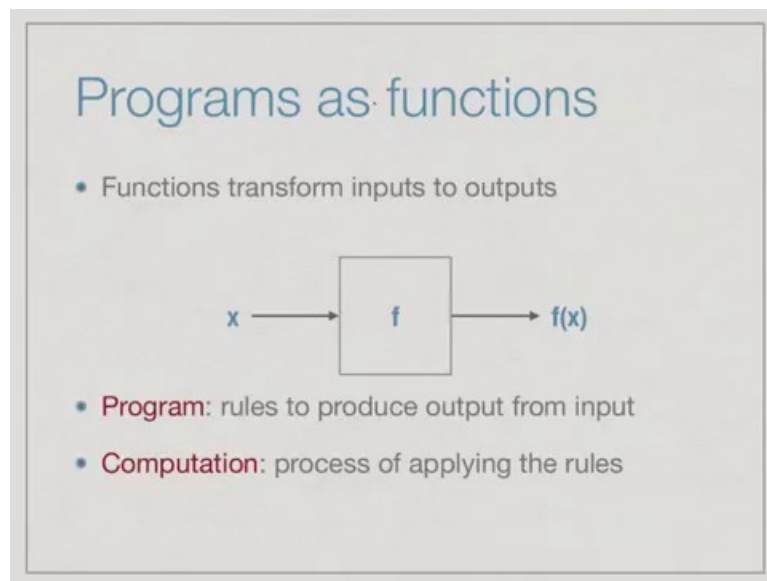
Chennai Mathematical Institute

Module # 01

Lecture – 02

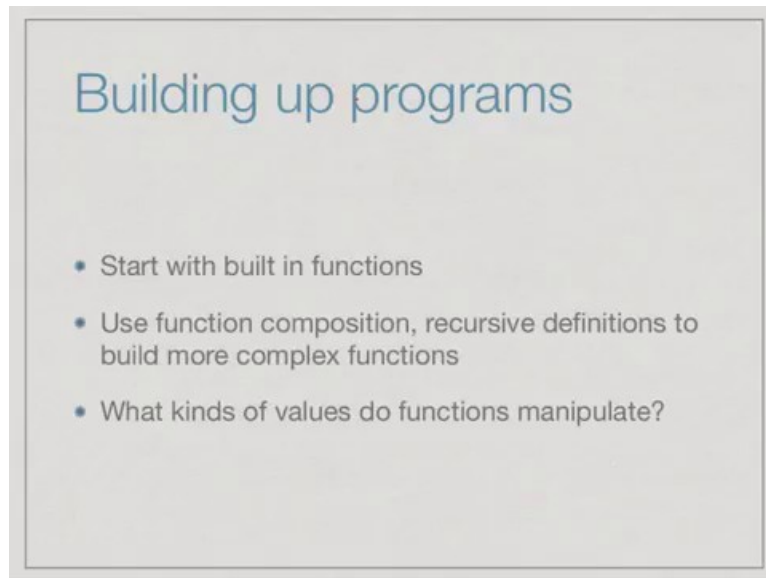
Types

(Refer Slide Time: 00:03)



In the previous lecture, we saw that we would like to look at programs as functions that transform inputs to outputs. So, our program is basically a set of rules that describes how to produce a given output from an input. And a computation consists of applying these rules and transforming the input to the output by repeatedly rewriting expressions based on the rules given.

(Refer Slide Time: 00:29)

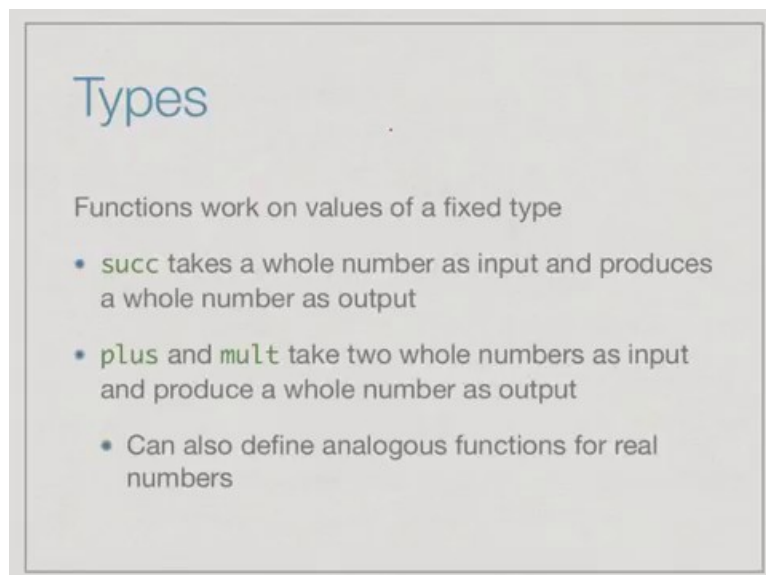


Building up programs

- Start with built in functions
- Use function composition, recursive definitions to build more complex functions
- What kinds of values do functions manipulate?

So, the way we write a program is to start with some built in functions and values. And then, we use function composition and things like recursive definitions to build more complex functions from the basic built in functions and values given to us. So, another thing that we need to be aware of when we are dealing with functions is to look at the values that they manipulate, so these are what we call data types or just types.

(Refer Slide Time: 00:59)



Types

Functions work on values of a fixed type

- `succ` takes a whole number as input and produces a whole number as output
- `plus` and `mult` take two whole numbers as input and produce a whole number as output
- Can also define analogous functions for real numbers

So, functions are defined over fixed input and output types. For instance, we saw last time the function successor named `succ` which adds one to the input. Now, the successor function that

we defined assumes that inputs are the whole numbers 0 1 2 3 and as output it produces another whole number. We also looked at definitions of plus and multiply. So, plus and multiply each took two whole numbers as input and produced a whole number as output. Notice that if you add two whole numbers or you multiply two whole numbers you will necessarily get a whole number as an output. Now, you could also define similar functions as plus and multiply for values that are not whole numbers or even successor.

So, plus 1 could be defined for any value, but successor means something. Successor means the next number. So, it makes sense to say that the next whole number after 1 is 2, but it does not make sense to say that the next fractional number after 1.5 is 2.5. There is no next fractional number, which is why successor is usually defined only for whole numbers. But, if you think of successor not as plus 1, then we can define plus and multiply, the general addition even for fractional numbers, but these will be different functions.

(Refer Slide Time: 02:16)

The slide is titled "Types" in blue. To the right of the title, there are handwritten blue equations: $\sqrt{4} = 2$ and $\sqrt{2} = 1.414\dots$. Below the title, the text "How about `sqrt`, the square root function?" is written. Underneath this, there is a bulleted list with three items. At the bottom of the slide, the mathematical expression $\mathbb{Z} \subseteq \mathbb{R}$ is written in red.

Types

$\sqrt{4} = 2$
 $\sqrt{2} = 1.414\dots$

How about `sqrt`, the square root function?

- Even if the input is a whole number, the output need not be—may have a fractional part
- Number with fractional values are a different type from whole numbers
- In Mathematics, whole numbers are often treated as a subset of fractional or real numbers

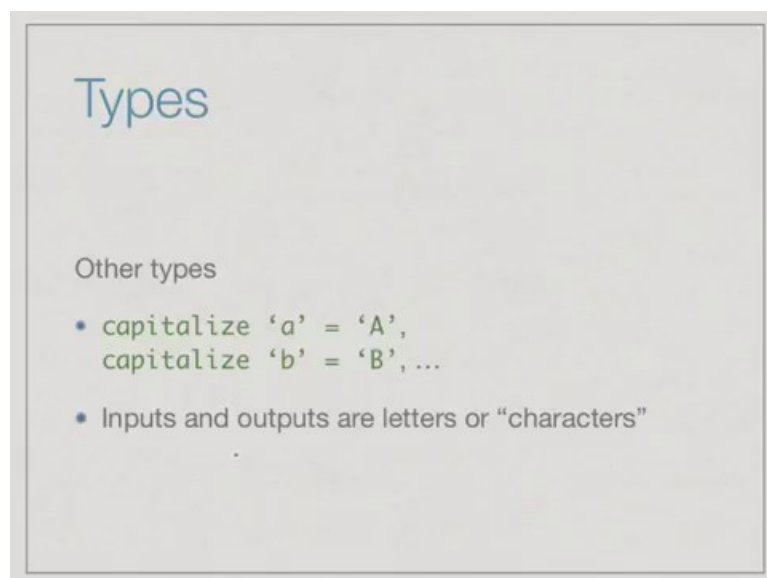
$\mathbb{Z} \subseteq \mathbb{R}$

So, what about a function like square root that takes the square root of a number. So, here, notice that even if the input is a whole number, the output need not be. The square root of 4 is 2, but the square root of 2 is in fact an irrational number, but at the very least it is the fractional number which is neither 1 nor 2. So, we have two different types of numbers that we deal with, we deal with these whole numbers or integers and we also deal with numbers which have fractional components.

The problem with fractional components is that the value after the decimal point can be arbitrarily long, it could be infinite in general. So, to precisely describe a fractional value, the description requires an infinite amount of information. In mathematics, typically, you write sets like \mathbb{R} for the real numbers and \mathbb{Z} for the integers and you assume implicitly that the integers are those real numbers which have a 0 fractional part.

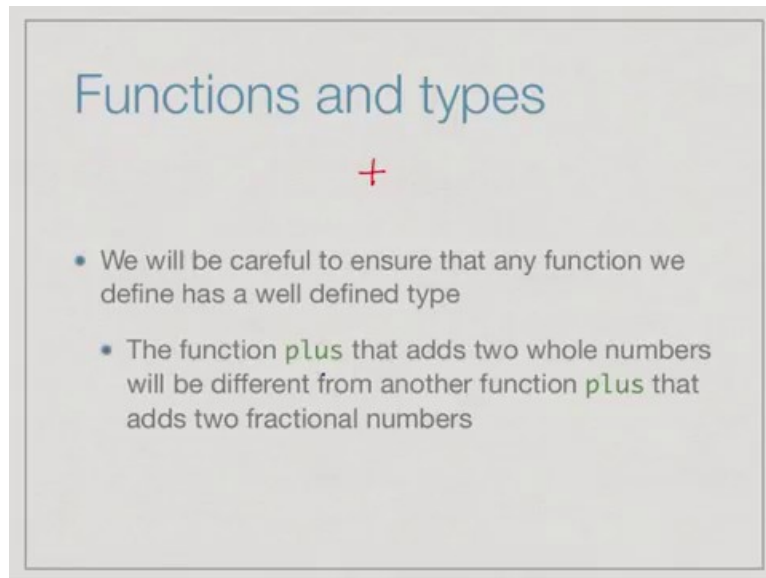
Now, when we write programs it turns out that representing integers and representing fractional numbers are very different and therefore, these are distinct types also. So, we will always differentiate between whole numbers and fractional numbers when we write functional programs, even though mathematically one is often seen as a subset within the other.

(Refer Slide Time: 03:40)



Now, many interesting programs manipulate values, which are not numbers. For instance, when you use a text editor or you enter an url in a browser, you are typing some text and then the program that is the web browser or your text editor has to interpret this text. So, characters are a very important type in programming. So, think of very simple function that we can write, something that takes an input letter and provided it is a valid letter of the alphabet not a punctuation mark or a number, it capitalizes it. So, capitalize 'a' is capital A, capitalize 'b' is capital B and so on. So for such a function the input and the output consists of characters.

(Refer Slide Time: 04:22)



The slide is titled "Functions and types" in a blue font. Below the title is a red plus sign. There are two bullet points in the slide:

- We will be careful to ensure that any function we define has a well defined type
- The function **plus** that adds two whole numbers will be different from another function **plus** that adds two fractional numbers

So, whenever we write functions in a functional programming language we will have to ensure that the functions' type is well defined. In fact, it will turn out that the functional language we are going to use, which is Haskell will not allow us to write a function whose type is not defined. Now, not defined can mean many things, but one of the things it means is that the output must be uniform. So, you cannot have a function, which on some values produces an integer as an output and on some other values produces a character as an output.

So, this will be the kind of limitation that we will see. So all functions will have to have a well defined type. And therefore, the same function which say mathematically does a same thing like plus, which mathematically does a same thing for whole numbers and for fractional numbers will actually consist of two different functions. Now it may be that for convenience we use a symbol like '+' or the name 'plus' to define both functions. But, in practice, actually, internally they are two different functions, because they have different types.

(Refer Slide Time: 05:23)

Functions have types

$\text{plusTwo } n = \text{succ } (\text{succ } n)$
 $\text{applyTwice } f\ n = f\ (f\ n)$

- A function that takes inputs of type **A** and produces output of type **B** has a type **$A \rightarrow B$**
- In Mathematics, we write **$f: S \rightarrow T$** for a function with domain **S** and codomain **T** $\text{range}(f) \subseteq T$
- A type is just a set of permissible values, so this is equivalent to providing the type of **f**

So, now, not only do the inputs and the outputs of functions have types, the functions themselves have types. We will see that we can write functions, which take other functions as input and manipulate them. For instance we can think of a function, ok we saw that, for instance, `plusTwo n` is `succ (succ n)`.

Now in general, we could say why to fix the number of times we apply successor. Instead, why not we say that the successor has to be applied twice. So, we take as input a function and a number and we say that the output is the result obtained when you are applying the function twice to that number. So, here notice that the function itself is being supplied to our new function and it is being used twice in the output. So, there is nothing that prevents us from writing functions, which actually take other functions as arguments. And now, we assumed that all functions must have well defined types for inputs and outputs, so functions must also have types.

Now, the notion of a function type is not very new to us. In mathematics, if we write a function from a set **S** to a set **T** we write **$f: S \rightarrow T$** . This means inputs come from **S**. So, **S** is often called the domain of the function **f** and outputs are from the set **T**. So, **T** is often called the co domain. In particular, the range of the function is that subset of the co domain, which is actually reached by values from **S**. So, we have this notion of an input set and an output set.

So, a type of a function in a programming language is exactly that. It just takes the input type and the output type and says it transforms the values of type **A** to values of type **B**. So, a type

is just a set of values. When we say we have the set of whole numbers, then it just means that the values permitted are 0 1 2 3. We say we have set of characters when we have a set of symbols that we are allowed to type. When we say fractional numbers, we have certain values that we can define with decimal points. So, a type is just a set of values and just like in a mathematical notation we write a function from one set to another, we use the same notation to denote the type of function and this is important, because one function can be actually fitted to another function as input as we will see.

(Refer Slide Time: 07:48)

Collections

Whole number 0,1,2.
Set of whole numbers {1,3,7}

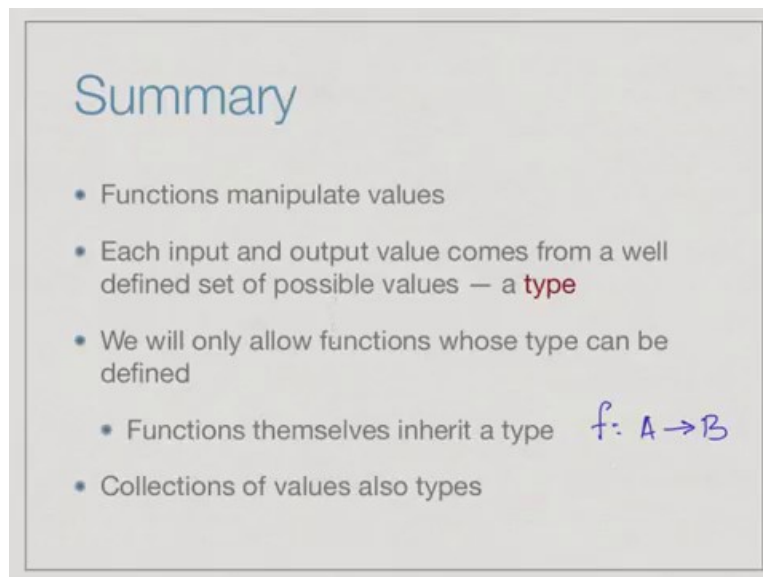
- It is often convenient to deal with collections of values of a given type
 - A list of integers
 - A sequence of characters — words or strings
 - Pairs of numbers (7.3, 4.2)
 - Such collections are also types of values

The other interesting notion is that of building collections of values. So, if you have whole numbers then we have individual values 0 1 2... Now you can have sets of whole numbers. For example in mathematics, a set of whole numbers is a collection of whole numbers say 1 3 7 and we write it with some notation to indicate that this is a collection, may be curly brackets and commas.

So, similarly in programming we often want to deal with collections of values. We might want to talk about a sequence or a list of integers. Or, when we type words, then a word consists of a number of characters, so that is a sequence of characters. Or, we might talk about points in space, x y coordinates. So, an x y coordinate is just a pair of values like 7.3 and 4.2, so we want pairs of numbers. So, if we have types for the components, then we have a type for the whole, so we can say that I have a pair of fractional numbers, I have a sequence of integers or I have a sequence of characters.

So, collections are also important types. So, a type remember, now just a set of permitted values. So, if I tell you the type then you can tell me whether a given value is legal for the type or not. If I tell you that the type is whole numbers and I ask you whether seven point five is a whole number you can say that it is not a whole number. If I ask you whether the list consisting of 1 followed by 2 is a list of integers ,you say yes it is the list of integers and so on.

(Refer Slide Time: 09:26)



Summary

- Functions manipulate values
- Each input and output value comes from a well defined set of possible values — a **type**
- We will only allow functions whose type can be defined
 - Functions themselves inherit a type $f: A \rightarrow B$
- Collections of values also types

So to summarize, we are talking about functions that manipulate values, but the values must come from well specified sets or types. So, each input and output value comes from a well defined set of possible values and we will only allow function definitions whose type is precisely specified. And given the types of the input and the output we saw that the function itself will have a type, which is from the input type to the output type.

And finally, we saw that it is important to have collections of values, so lists of integers or pairs of characters. So, we want to combine individual values or smaller values into larger values, which can store multiple values of the same type.