Function Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh
Chennai Mathematical Institute

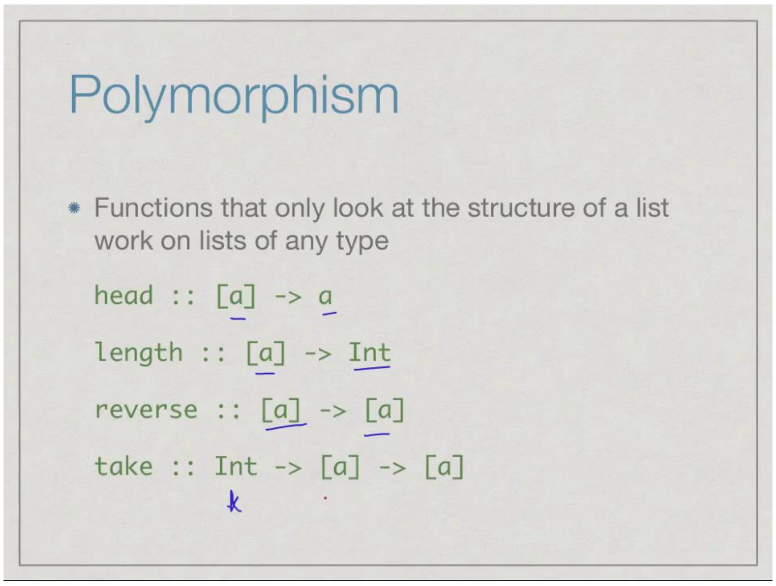
Module # 04

Lecture – 04

Conditional Polymorphism

Let us investigate polymorphism in Haskell a little more closely.

(Refer Slide Time: 00:07)



Polymorphism

- Functions that only look at the structure of a list work on lists of any type

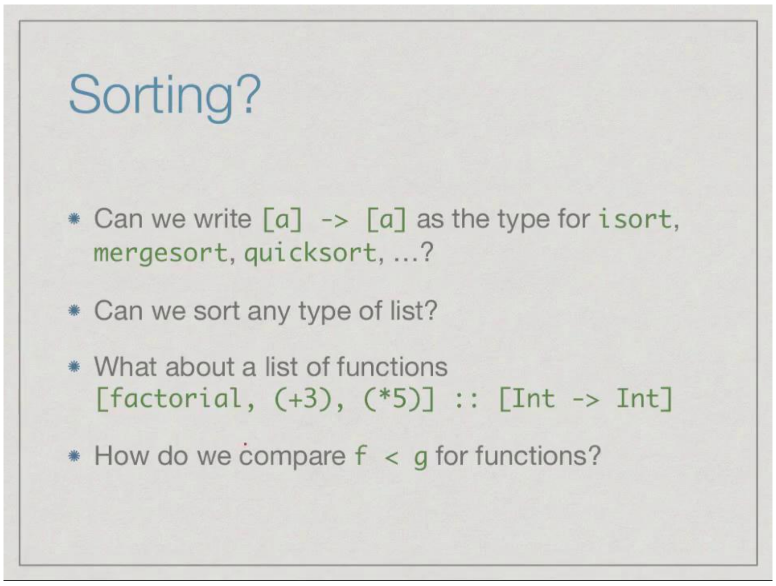
```
head :: [a] -> a
length :: [a] -> Int
reverse :: [a] -> [a]
take :: Int -> [a] -> [a]
```

What we have seen is that for functions that only look at the structure of a list, we can make them work on list of any type. So, for instance, the head function takes a list which is non-empty and removes the first element, it does not really care, what the value of the first element is, it just returns it. So, remember we should think of this in terms of boxes.

So, supposing we think of a list as a collection of boxes arranged in a row and somebody, ask you for the first box, you can give it to them without opening it. So, head we say takes a list of any type a and produces the value of type a. Similarly, length just counts the boxes, so it walks down the list, a list of any type a and counts, how many boxes there are and returns an integer. Reverse just rearranges the boxes, so that the first is last and the last is first and finally, take for example, gives a number.

So, take generalizes in a sense head, you say not the first box, I want the first k boxes. So, you take a k, then you take a collection of boxes and remove the first k of them, but once again, you do not need to look inside the box.

(Refer Slide Time: 01:15)



Sorting?

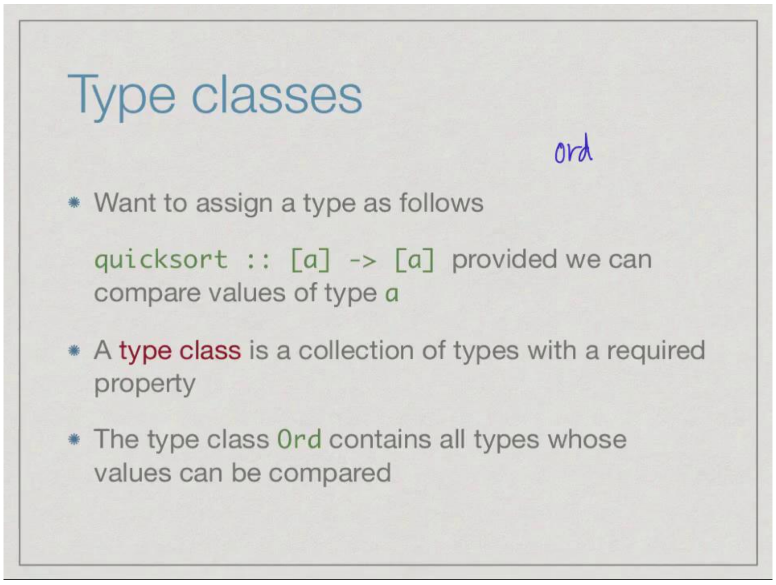
- Can we write `[a] -> [a]` as the type for `isort`, `mergesort`, `quicksort`, ...?
- Can we sort any type of list?
- What about a list of functions `[factorial, (+3), (*5)] :: [Int -> Int]`
- How do we compare `f < g` for functions?

So, what about the sorting functions which we saw, we saw insertions sort, merge sort, quick sort, can we legitimately say that these functions are of type list of a to list of a. Because, after all that take a list and produce a list and the way that you do, say insertions sort for integers is the same as for float and it is the same for many other types that you can think off.

So, why we cannot say this, so the question is, can we sort any type of list, which is what this type would suggest. You say that quick sort or insertion sort or merge sort will take any list of a and return back a list of same type. Now, the problem is that, list can contain any uniform type, just as we have seen that, we can pass functions to other functions, so there is no specific limitation on what types of objects, we can move around between functions. There is also no limitation on what types of objects, we can put into a list.

So, we can as well consist construct a list of functions. So, here is a list of functions, all of them have the type Int to Int, they take an Int as an integer, as an argument and produce some Int as a result. So, we have factorial which is of this type, we have plus 3, remember plus 3 is the function which adds 3 to whatever I give it. We have times 5, which you multiply by 5 whatever I give it. So, this is the legitimate list. Now, how would I say, whether or not the function factorial is less than the function plus 3, whether plus 3 is greater than or equal to the function multiplied by 5.

(Refer Slide Time: 02:51)



Type classes

ord

- Want to assign a type as follows

  `quicksort :: [a] -> [a]` provided we can compare values of type *a*

- A type class is a collection of types with a required property

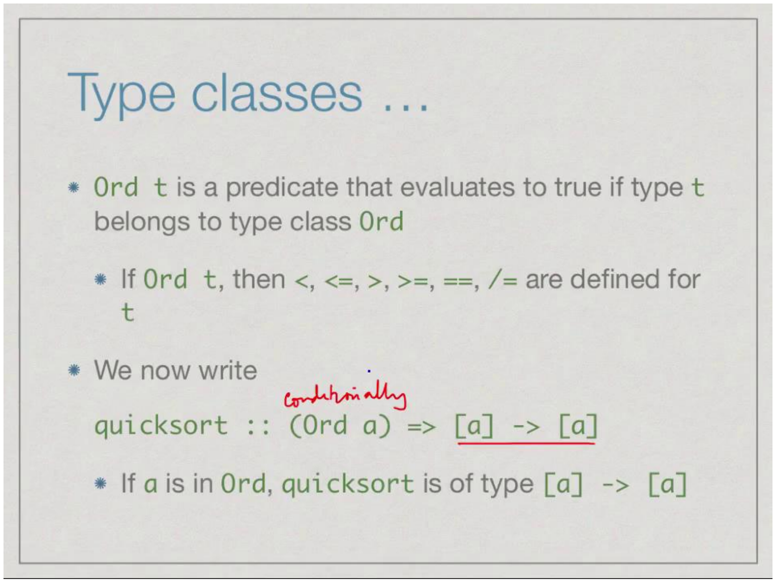
- The type class `Ord` contains all types whose values can be compared

So, what web to want say is that, we want to restrict the type of function like quick sort to all those list, from list a to list a, provided we can compare values of type a. So, you want to rule of things like these functions, which we do not have a sensible way of compare. So, in Haskell, this is captured using a notion called a type class. So, a type class is a collection of types with a require property.

In this case, the type class which is relevant to us is the type class called Ord, notice the capital letter O. So, type class is usually written with a capital letter and Ord contains all types, whose values can be compared, this includes since like integers and other number types like float. It also includes since like Boolean, where false is determined be less is defined to be less than 2.

We have characters which can be ordered depending on their internal representations. So, whatever value ord, the small ord the function on characters return, so this allows ordered the characters and so on. So, capital Ord set of all types, whose values can be compared.

(Refer Slide Time: 04:03)



Type classes …

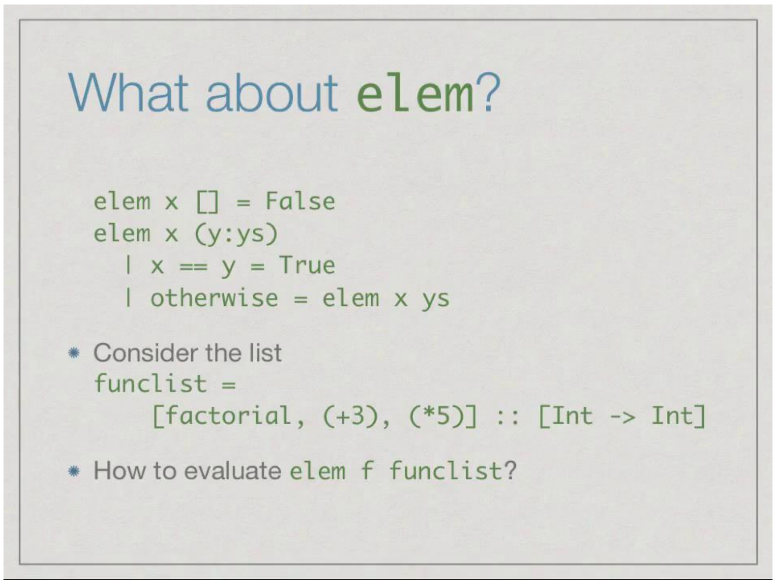
- Ord t is a predicate that evaluates to true if type t belongs to type class Ord
  - If Ord t, then <, <=, >, >=, ==, /= are defined for t
- We now write

  quicksort :: $\overset{\text{conditionally}}{\text{(Ord a)}}$ => $\underline{\text{[a] -> [a]}}$
  - If a is in Ord, quicksort is of type [a] -> [a]

So, we can think of capital Ord as predicate which evaluates to true on a type t, if t belongs to Ord, it is not actually predicate, but let us just think about at this way. So, what internally it means is that, if t belongs to the class Ord, then the comparison function is less than equal to, equal to, where not equal to etcetera, defined for t, which internal allows us to sort these allowance. Because, these are the values, these are the functions used to compare values of this type.

So, now, we can write the type of quick sort in this way, it is a kind of conditional polymorphism. So, this part says that, it is from list of a to list of a, but is not from any list of a to list of a, it must be a list, whose elements can be compared. So, there is a new symbol here is you should read as a kind of logical implication. So, this cells, if the underline type a belongs to Ord, then quick sort is of type list of a to list of a. So, this is not unconditionally list of a to list of a, this is conditionally list of a to list of a provided a is an Ord type.

(Refer Slide Time: 05:18)



What about `elem`?

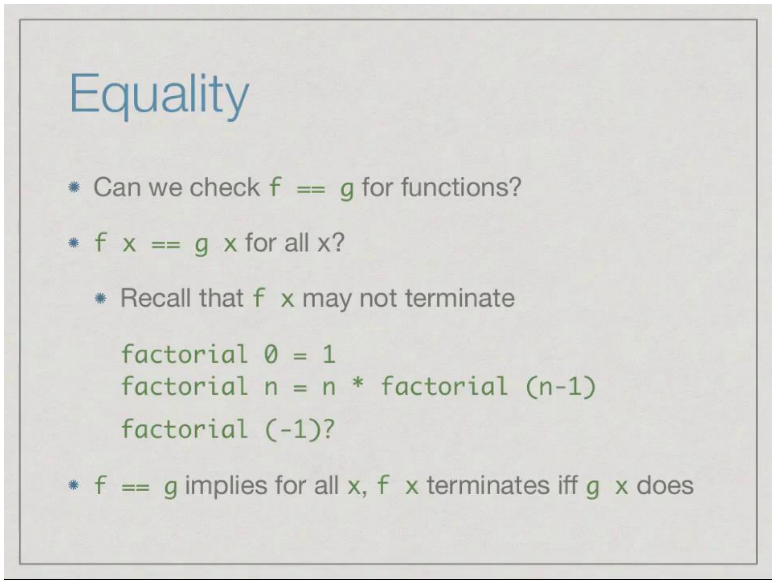
```
elem x [] = False
elem x (y:ys)
  | x == y = True
  | otherwise = elem x ys
```

- Consider the list
  ```
  funclist =
      [factorial, (+3), (*5)] :: [Int -> Int]
  ```
- How to evaluate `elem f funclist`?

So, this make sense for sorting, what about the more basic function we have seen the function elem, we checks whether a value belongs to a list of a. So, inductively we said that elem of x with an empty list is always false and we have a non-empty list, check the first value, if it is equal, then say true, otherwise, then say false, I am not say false, otherwise check the ((Refer Time: 05:44)).

So, here what we are really doing is, checking whether a given value is equal to another value. Now, this seems ((Refer Time: 05:53)) it could make sense seems to check whether two values are the same or not. Again our problem comes, because our values include the so called higher order types, namely functions. So, let us look at our old list that we had before which you have problem sorting. So, now, the question is, what does it mean to ask, whether some other function f belongs to this list or not.

(Refer Slide Time: 06:19)



Equality

- Can we check f == g for functions?
- f x == g x for all x?
  - Recall that f x may not terminate

```
factorial 0 = 1
factorial n = n * factorial (n-1)

factorial (-1)?
```

- f == g implies for all x, f x terminates iff g x does

We need to be able to check, whether f one function is equal to g an other function, is this are easy to think to do. So, one way of course to answer this question is to check, whether the value of the function f is the same as the value of the function g for every x. Therefore, in principle, there must be of the same type and they must agree on all the outputs. So, this is sometimes called the extensionality principle.

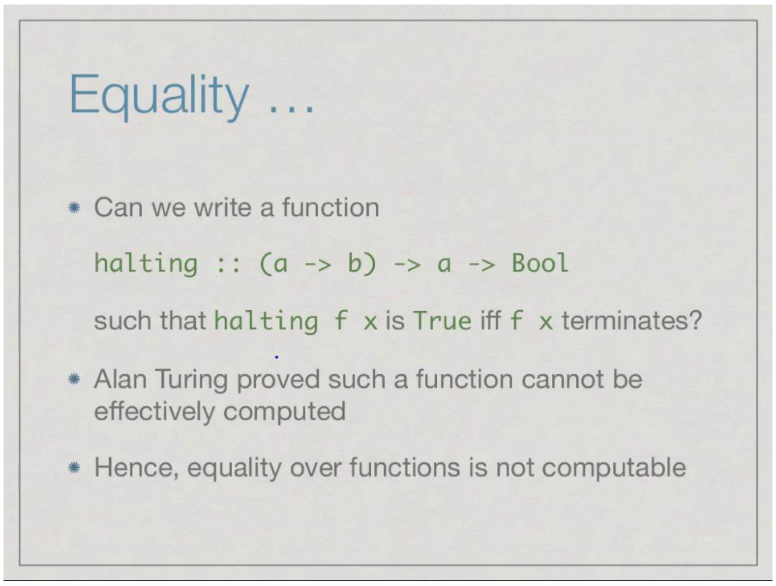
So, it would say for instance for any correct sorting algorithm is equal to any other correct sorting algorithm, because both of them and given an input list of the same arrangement, will produce an output list with the same arrangement. So, we were ignoring things like efficiency another characteristics, we are just saying, the input output ((Refer Time: 07:06)) function are same, so can we not do this.

So, other than the minor problem that have to check this for an infinite set of values, there is also the more serious problem that computations do not always terminate. Remember, our initial light declaration of factorial which ignores the problem of negative inputs. So, if he had a function factorial definition like this, we did not take care of negative inputs carefully, then the value of factorial are something like minus 1, would degenerate into an entire computation that it call factorial minus 2 and then factorial minus 3 and so on.

And therefore, this recursive step would never terminate. So, we would never get an answer. So, either if you could in some systematic way of check, f of x equal to g of x for all x at the very least, we need to be end to check on the way that f of x is not only give as sensible value. So, maybe we want to say it, whenever say f of minus 1 is not terminating, g of minus 1 is not terminating.

So, in particular, we need to be able to check at least this much for get more checking the values, we need to at least check the term terminate on the same sets of inputs. So, not just that we are not weakening our condition, we are not checking that f of x is equal to g of x, we are just asking does f of x terminate exactly when g of x terminates.

(Refer Slide Time: 08:30)



Equality …

- Can we write a function

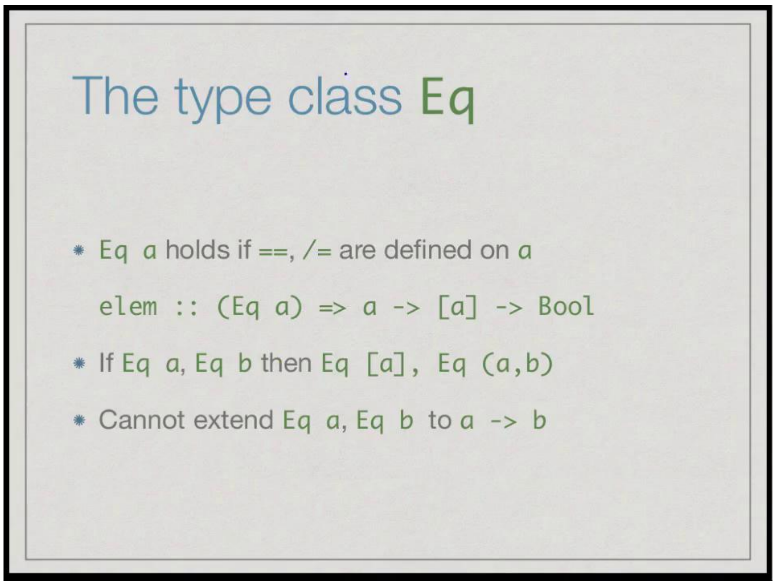
  `halting :: (a -> b) -> a -> Bool`

  such that `halting f x` is `True` iff `f x` terminates?
- Alan Turing proved such a function cannot be effectively computed
- Hence, equality over functions is not computable

So, in other words, we need to a function such as halting which will take the function, take an input and tell us, yes, f terminates when evaluated on x. So, halting f of x is true, provided f of x terminates false otherwise. So, this is the minimum that we need, in order to email address the issue of how to compare two functions and what Alan Turing proved is that, this function cannot be computed.

So, this is his famous result about the halting problem, there is no algorithm which can determine whether an other program and then input to that program halts or not. So, given that we continent compute, when two functions agree on the terminating inputs, it is clear that we cannot compute any sensible notion or equality over functions. In other words, equal to equal to is not for basic fact for every type in our system.

(Refer Slide Time: 09:28)



The type class Eq

- Eq a holds if ==, /= are defined on a

  elem :: (Eq a) => a -> [a] -> Bool

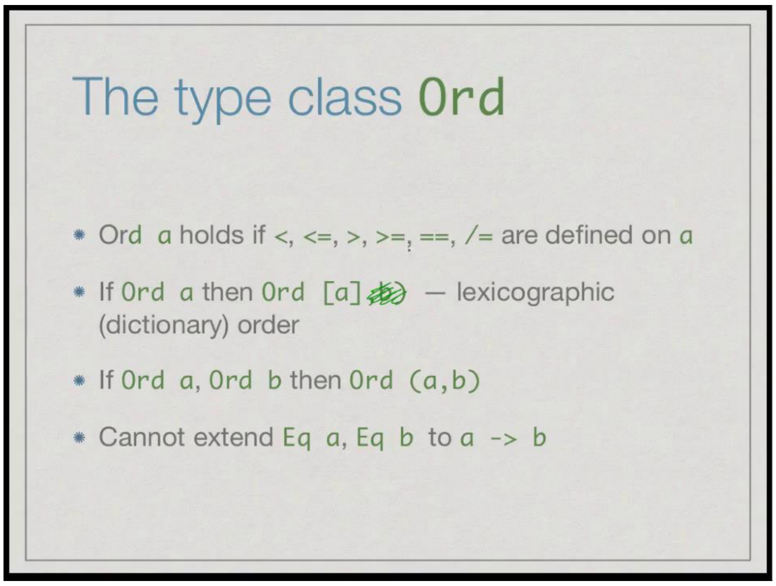
- If Eq a, Eq b then Eq [a], Eq (a,b)
- Cannot extend Eq a, Eq b to a -> b

So, since equality is not define our functions, we need to have a type class called Eq, which specifies at a given type as values at can be compared for equality and inequality. So, Eq of a holds provided equal to equal to and not equal to or define on values of that type. So, incomes of this type class, we can now give a conditionally polymorphic type for the elem function, which says a provided Eq holds for the input type. Then, we take a value of the type on the list of that type and tell whether or not the value occurs in that list.

So, fortunately all the built in types in Haskell starting with integers, floats, Char, Bool, etcetera, all support equality and if you have a list of values, each of which can compare to equality, they can compare the list for equality element to element. We can compare to element to element and so on. So, we can extend E q from the underline type to structure types.

On the other hand as we have seen, even though we might able to apply E q to the input and the output types separately for function, it does not mean that we have E q define or the function type, because there is no sensible way to compute equality of function in general.

(Refer Slide Time: 10:55)

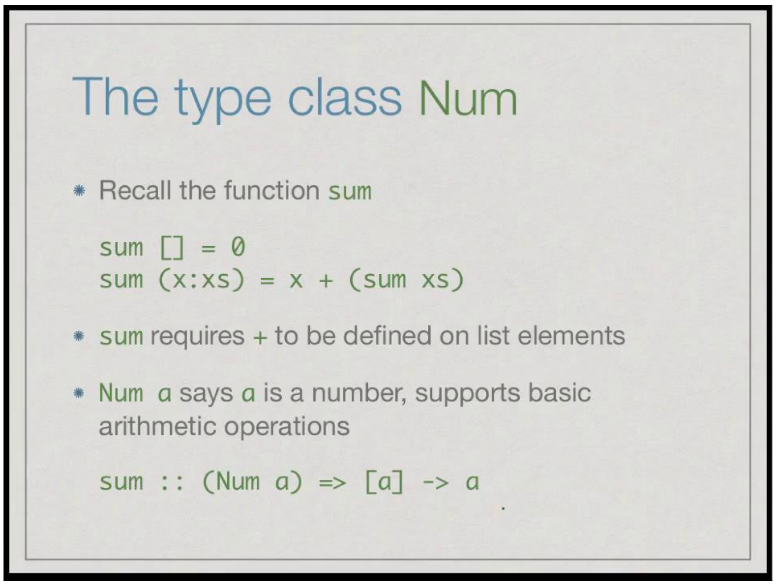


The type class Ord

- Ord a holds if <, <=, >, >=, ==, /= are defined on a
- If Ord a then Ord [a] — lexicographic (dictionary) order
- If Ord a, Ord b then Ord (a,b)
- Cannot extend Eq a, Eq b to a -> b

The same way remember that Ord or say holds, if the comparison functions are defined on values of the type. Now, if we have type a on which we can compare values, then we can compare list of those two values by using lexicographic or dictionary order. So, we just list out the values and look for the left most value is differs and based on the order of this left most differing value which is exactly how you look up words in the dictionary, we decide this list is smaller than the other. In the same way, we can also compare ((Refer Time: 11:34)) but in the same way, we cannot compare functions.

(Refer Slide Time: 11:40)



The type class Num

- Recall the function sum

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

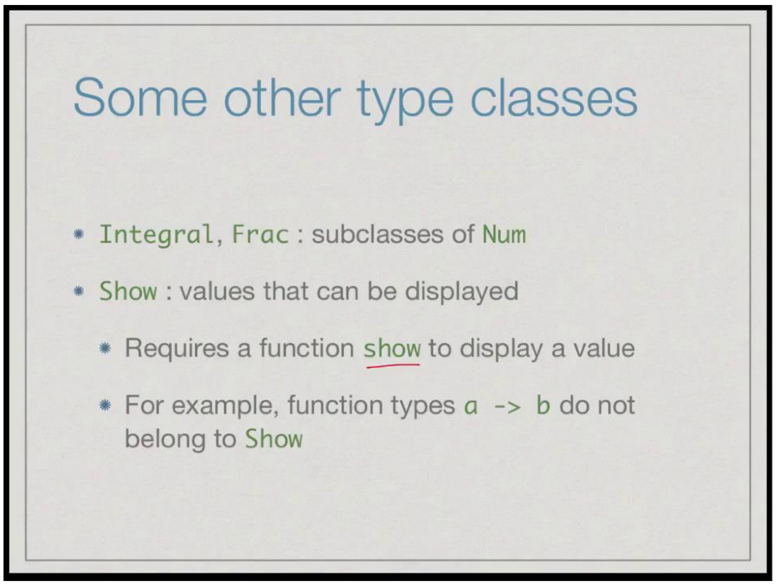
- sum requires + to be defined on list elements
- Num a says a is a number, supports basic arithmetic operations

```
sum :: (Num a) => [a] -> a
```

Another type, which is very common is the type class num, which tells us whether arithmetic operations OR and OUT. So, remember the functions sum, which compute sum of the list. So, sum of the list just takes 0 as the sum of negative list and adds a all the values inductive. So, sum requires the function plus to be defined on the values of the list. So, num a faces that a is a number that supports basic arithmetic operations.

So, the correct polymorphic type for sum is, if num of a, then given a list of a, it produces the value of the same type. Now, why of the same type, where obviously, if you are adding an integers, you will get an integer, if you adding up a floats, you will get float and so on. So, provide the underline values can be added, you will get a value of the same type.

(Refer Slide Time: 12:34)



Some other type classes

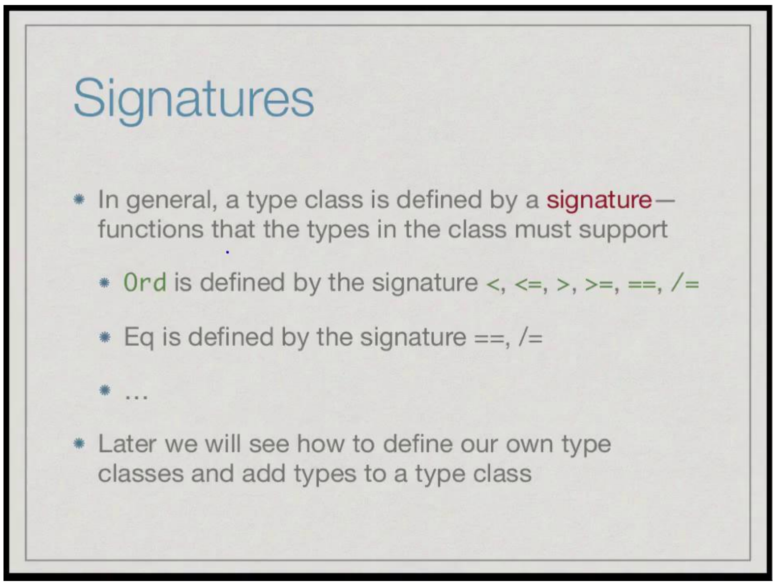
- Integral, Frac : subclasses of Num
- Show : values that can be displayed
  - Requires a function show to display a value
  - For example, function types a -> b do not belong to Show

So, num actually has specialized sub classes such as integral, Frac, etcetera. So, you can look up some Haskell documentation to see all of various type classes at are built in to Haskell. There is a very important class for show which tells us, when a value can be actually display to the user. So, it requires on function also called show with small s to be define.

So, again for all the basic types, we have a show function which gives us representation that we seem and we work with ghci. On the other hand, if you ask, Haskell to show what the value of the function is, then obviously, there is no sensible at to describe the function, because remember a function is a black box, it is an input output box.

So, we can possibly describe a function, it terms of all it is input output behaviors and display to the user, at the same time, that we may be many different viewers of describing a specific computational implementation of function. So, there is no sensible way to assign a function5

(Refer Slide Time: 13:38)



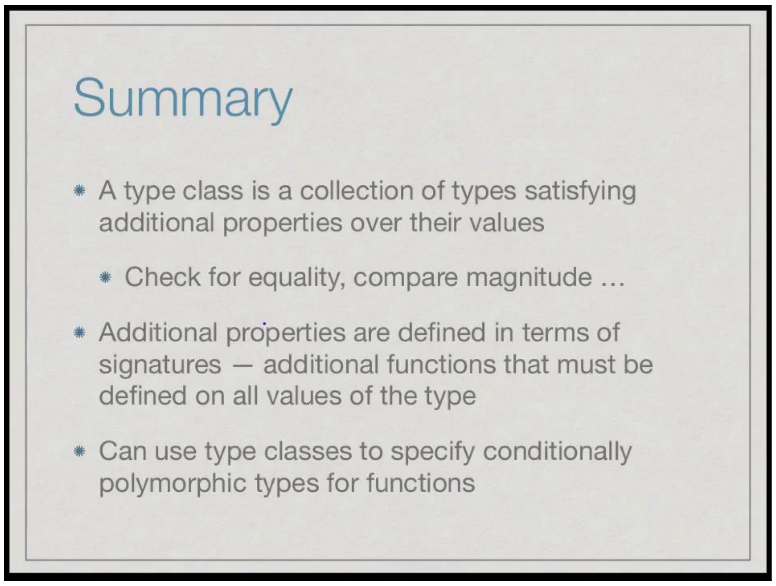
Signatures

- In general, a type class is defined by a signature—functions that the types in the class must support
  - `Ord` is defined by the signature <, <=, >, >=, ==, /=
  - Eq is defined by the signature ==, /=
  - ...
- Later we will see how to define our own type classes and add types to a type class

So, what we have seen is that, the way that Haskell deals with type classes is to specify them in terms of signatures. He says that, if you want to type to belong to Ord, you must if sensible definitions for less than equal to greater than. By sensibility nearly means at you must give a value of the correct type, it does not introduce any meaning to it, you can get a completely unnatural definition of less than or less than equal to and still claim that something belongs to Ord. I.

t says that thinks like sorting will not behave and the expected it. Similarly, any functions any type can be make a number of E q by providing the suitable definition for equal to and not equal to and so on. So, we will see how to do this later on, we will see and how to add or own type classes or to add types to a given type class and so on.

(Refer Slide Time: 14:29)



Summary

- A type class is a collection of types satisfying additional properties over their values
  - Check for equality, compare magnitude …
- Additional properties are defined in terms of signatures — additional functions that must be defined on all values of the type
- Can use type classes to specify conditionally polymorphic types for functions

So, to summarize a type class is a collection of types, let satisfies additional properties of the values. For instance, there will be to check for equality to compare values by magnitude and so on and these additional properties are defined in terms of signatures. So, these are additional functions that must be defined are all values of that type in order to use them as part of the type class. And then, once we have type classes, we can get conditionally polymorphic types for functions, such as sorting, which require the underline values, you have a certain property or for elem and so on.