

Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

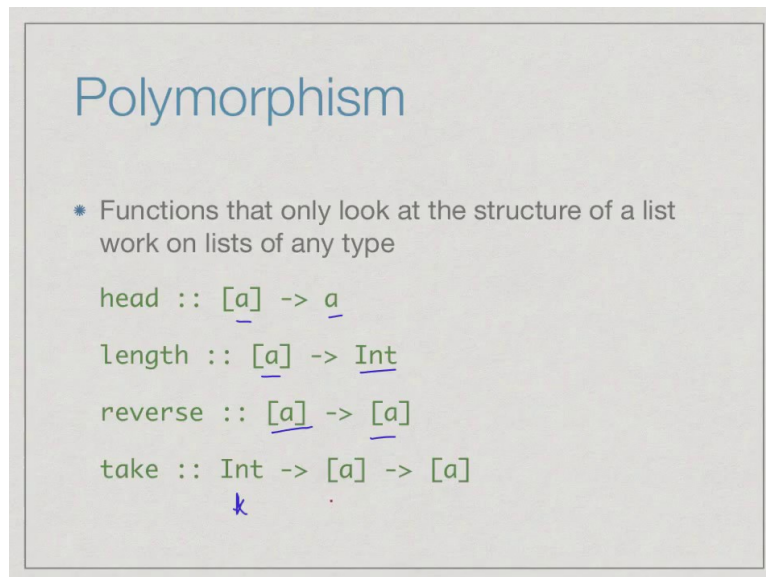
Module # 04

Lecture – 04

Conditional Polymorphism

Let us investigate polymorphism in Haskell a little more closely.

(Refer Slide Time: 00:07)



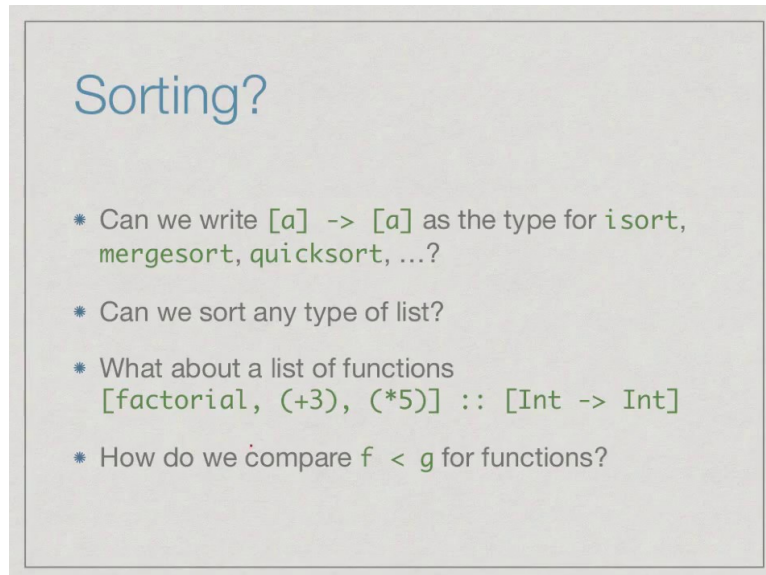
The slide is titled "Polymorphism" in a large blue font. Below the title, there is a bullet point: "• Functions that only look at the structure of a list work on lists of any type". Underneath this, four Haskell function signatures are listed in green text, with some parts underlined in blue. The signatures are: `head :: [a] -> a`, `length :: [a] -> Int`, `reverse :: [a] -> [a]`, and `take :: Int -> [a] -> [a]`. In the `take` signature, the `Int` is underlined in blue, and there is a small blue 'k' below it. There is also a small red dot below the second `[a]` in the `take` signature.

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 a 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 the function 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?

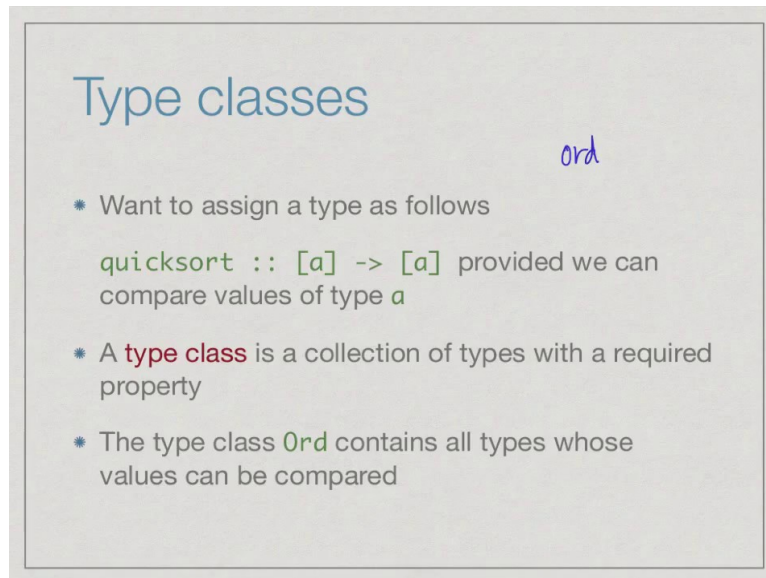
- * Can we write `[a] -> [a]` as the type for `isort`, `mergesort`, `quicksort`, ...?
- * Can we sort any type of list?
- * What about a list of functions
`[factorial, (+3), (*5)] :: [Int -> Int]`
- * How do we compare `f < g` for functions?

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

So, why can we not say this, so the question is, can we sort any type of list, which is what this type would suggest. It would say that quicksort or insertion sort or merge sort will take any list of a and return back a list of the 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 construct a list of functions. So, here is a list of functions, all of them have the type `Int -> Int`, they take an integer as an argument and produce some `Int` as a result. So, we have `factorial` which is of this type, we have `(+3)` remember `(+3)` is the function which adds 3 to whatever I give it. We have `(*5)`, which will 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 `(+3)`, whether `(+3)` is greater than or equal to the function `(*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 we want to say is that, we want to restrict the type of function like quicksort to all those list, from `[a] -> [a]`, provided we can compare values of type `a`. So, you want to rule out things like these functions, which 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 required 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 things like integers and other number types like float. It also includes things like Boolean, where `False` is defined to be less than `True`.

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 us to order the characters and so on. So, capital `Ord` is a 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 :: (Ord a) => [a] -> [a]`
conditionally
- * If `a` is in `Ord`, `quicksort` is of type `[a] -> [a]`

So, we can think of capital `Ord` as a predicate which evaluates to true on a type `t`, if `t` belongs to `Ord`, it is not actually a predicate, but let us just think about it this way. So, what internally it means is that, if `t` belongs to the class `Ord`, then the comparison function is `<`, `<=`, `==`, `/=`, etc defined for `t`, which in turn allows us to sort these elements. Because, these are the functions we use to compare values of this type.

So, now, we can write the type of `quicksort` in this way, it is a kind of conditional polymorphism. So, this part says that, it is from `[a] -> [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 which you should read as a kind of logical implication. So, this says if the underlying type `a` belongs to `Ord`, then `quicksort` is of type `[a] -> [a]`. So, this is not unconditionally `[a] -> [a]`, this is conditionally `[a] -> [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 makes sense for sorting, what about the more basic function we have seen the function elem, it checks whether a value belongs to a list or not. So, inductively we said that elem of x with an empty list is always false and if we have a non-empty list, check the first value, if it is equal, then say true, otherwise, check the rest of the list (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 innocuous enough ((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 when we had 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 another function, is this a reasonable thing 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, they 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 that any correct sorting algorithm is equal to any other correct sorting algorithm, because both of them when given an input list of the same arrangement, will produce an output list with the same arrangement. So, we are ignoring things like efficiency and other characteristics, we are just saying, the input output behaviour of the ((Refer Time: 07:06)) function are same, so can we not do this.

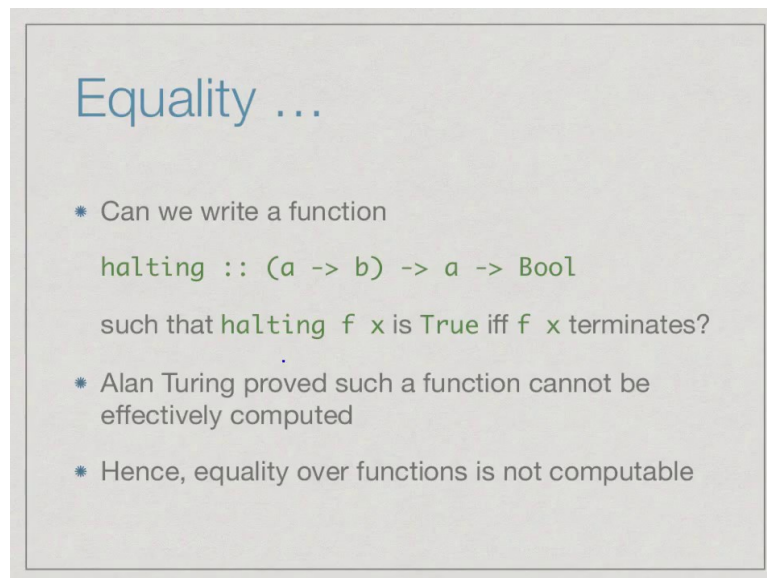
So, other than the minor problem that we have to check this for an infinite set of values, there is also the more serious problem that computations do not always terminate. Remember, our initial light declaration of factorial which ignored the problem of negative inputs. So, if we had a function factorial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

And therefore, this recursive step would never terminate. So, we would never get an answer. So, even if you could in some systematic way check, $f(x)$ equal to $g(x)$ for all x at the very least, we need to be able to check on the way that $f(x)$ is not going to give us a sensible value.

So, maybe we will want to say it, whenever say $f(-1)$ is not terminating, $g(-1)$ is not terminating.

So, in particular, we need to be able to check at least this much ,forget about checking the values, we need to at least check that they terminate on the same sets of inputs. So, not just that we are not weakening our condition, we are not checking that $f\ x == g\ x$, we are just asking does $f\ x$ terminate exactly when $g\ x$ 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 write 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 x` is true, provided $f\ x$ terminates false otherwise. So, this is the minimum that we need, in order to even 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 another program and an input to that program halts or not. So, given that we can't even compute, when two functions agree on their terminating inputs, it is clear that we cannot compute any sensible notion of equality over functions. In other words, equal to is not a 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 defined on functions, we need to have a type class called `Eq`, which specifies that a given type has values that can be compared for equality and inequality. So, `Eq a` holds provided `==` and `/=` are defined on values of that type. So, in comes of this type class, we can now give a conditionally polymorphic type for the `elem` function, which says that provided `Eq` holds for the input type, then, we can take a value of that type and 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`, etc, all support equality and if you have a list of values, each of which can compare to a equality, we can compare the list for equality element to element. We can compare tuples element to element and so on. So, we can extend `Eq` from the underlying type to structured types.

On the other hand as we have seen, even though we might be able to apply `Eq` to the input and the output types separately for function, it does not mean that we have `Eq` defined on the function type, because there is no sensible way to compute equality of functions 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`

In the same way remember that `Ord a` holds, if the comparison functions are defined on values of that type. Now, if we have a 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 which 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 which list is smaller than the other. In the same way, we can also compare tuples ((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`

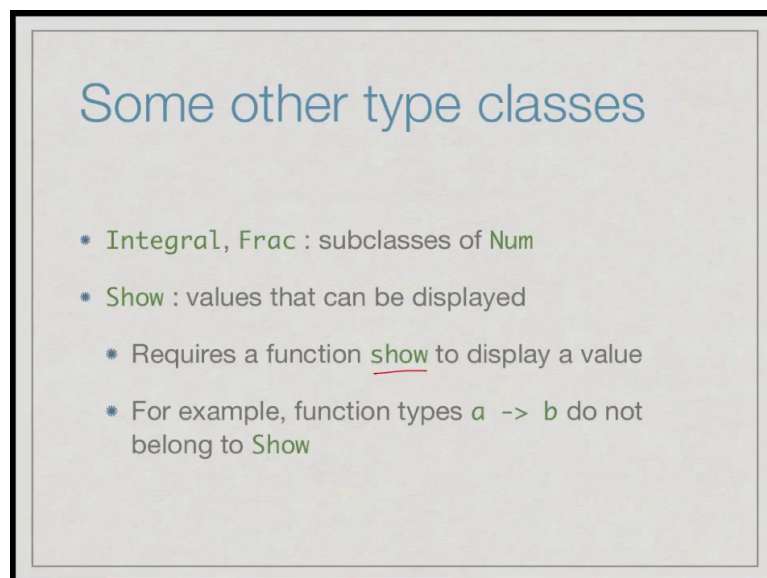
```
sum [] = 0  
sum (x:xs) = x + (sum xs)
```
- `sum` requires `+` to be defined on list elements
- `Num a` says `a` is a number, supports basic arithmetic operations

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

Another type, which is very common is the type class `Num`, which tells us whether arithmetic operations are allowed. So, remember the function `sum`, which compute sum of the list. So, `sum` of the list just takes 0 as the sum of negative list and adds up all the values inductively. So, `sum` requires the function `+` to be defined on the values of the list. So, `Num` says 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 `[a]`, it produces the value of the same type. Now, why of the same type, where obviously, if you are adding up integers, you will get an integer, if you are adding up floats, you will get float and so on. So, provided the underlying values can be added, you will get a value of the same type.

(Refer Slide Time: 12:34)

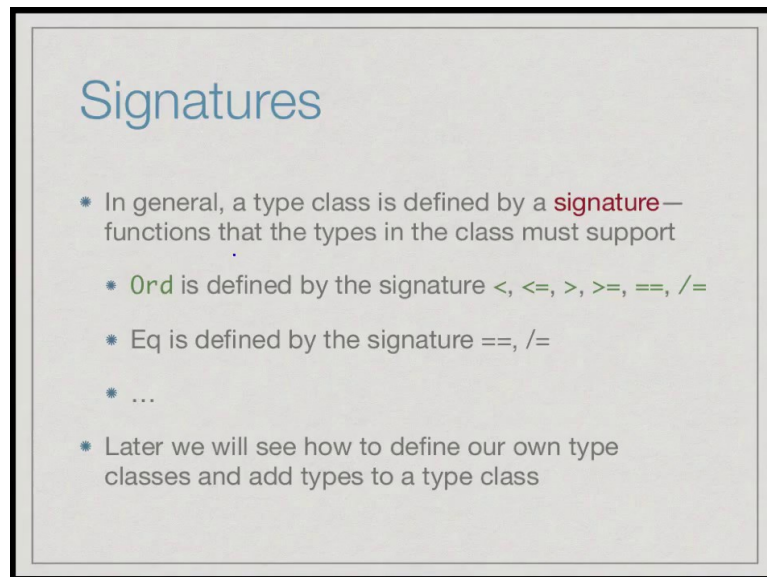


So, `Num` actually has specialized sub classes such as `Integral`, `Frac`, etc. So, you can look up some Haskell documentation to see all the various type classes that are built in to Haskell. There is a very important class called `Show` which tells us, when a value can be actually displayed to the user. So, it requires a function also called `show` with small `s` to be defined.

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

So, we can't possibly describe a function, in terms of all its input output behaviors and display it to the user, at the same time, there may be many different ways of describing a specific computational implementation of a function. So, there is no sensible way to assign a function type to the type class Show.

(Refer Slide Time: 13:38)



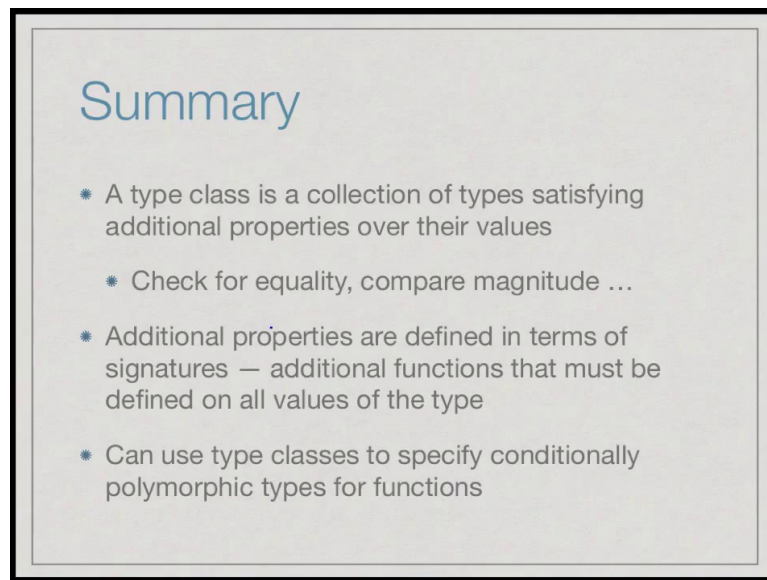
Signatures

- 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. It says that, if you want a type to belong to `Ord`, you must give sensible definitions for `<`, `<=`, `>`.. By sensible it merely means that you must give a value of the correct type, it does not introduce any meaning to it, you can give a completely unnatural definition of `<` or `<=` and still claim that something belongs to `Ord`.

It is just that things like sorting will not behave in the expected way. Similarly, any function any type can be made a member of `Eq` by providing the suitable definition for `=` and `\=` and so on. So, we will see how to do this later on, we will see how to add our 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, that satisfies additional properties over their values. For instance, the ability 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 on 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 give conditionally polymorphic types for functions, such as sorting, which require the underlying values to have a certain property or for elem and so on.