Programming in Haskell: Lecture 12

S P Suresh

September 12, 2019

Polymorphism

- Same code that works for objects of different types

Polymorphism

- Same code that works for objects of different types
- **Example**: Functions that only look at the structure of a list work on lists of any type

Polymorphism

- Same code that works for objects of different types
- Example: Functions that only look at the structure of a list work on lists of any type
- Example functions:

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

Sorting?

- Can we write `[a] -> [a]` as the type for `isort`, `mergesort`, `quicksort`, …?

Sorting?

- Can we write `[a] -> [a]` as the type for `isort`, `mergesort`, `quicksort`, …?
- Can we sort any type of list?

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]
```

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?

Type classes

- Want to assign a type as follows:

Type classes

- Want to assign a type as follows:
- `quicksort :: [a] -> [a]` provided we can compare values of type `a`

Type classes

- 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

Type classes

- 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

Type classes

- `Ord t` is a predicate that evaluates to `True` if the type `t` belongs to type class `Ord`

Type classes

- `Ord t` is a predicate that evaluates to `True` if the type `t` belongs to type class `Ord`
- Terminology: There is an `Ord` instance of type `t`

Type classes

- `Ord t` is a predicate that evaluates to `True` if the type `t` belongs to type class `Ord`
- Terminology: There is an `Ord` instance of type `t`
- Alternatively: `t` is an instance of `Ord`

Type classes

- `Ord t` is a predicate that evaluates to `True` if the type `t` belongs to type class `Ord`
- Terminology: There is an `Ord` instance of type `t`
- Alternatively: `t` is an instance of `Ord`
- If `t` is an instance of `Ord`, then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `t`

Type classes

- `Ord t` is a predicate that evaluates to `True` if the type `t` belongs to type class `Ord`
- Terminology: There is an `Ord` instance of type `t`
- Alternatively: `t` is an instance of `Ord`
- If `t` is an instance of `Ord`, then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `t`
- For `t` to be an instance of `Ord`, it should also be an instance of `Eq`

Type classes

- If `t` is an instance of `Eq`, then `==` and `/=` are defined for `t`

Type classes

- If `t` is an instance of `Eq`, then `==` and `/=` are defined for `t`
- If `t` is an instance of `Ord`, then `<`, `<=`, `>`, `>=` are also defined for `t` (in addition to `==` and `/=`)

Type classes

- If `t` is an instance of `Eq`, then `==` and `/=` are defined for `t`
- If `t` is an instance of `Ord`, then `<`, `<=`, `>`, `>=` are also defined for `t` (in addition to `==` and `/=`)
- Back to sorting …

Type classes

- If `t` is an instance of `Eq`, then `==` and `/=` are defined for `t`
- If `t` is an instance of `Ord`, then `<`, `<=`, `>`, `>=` are also defined for `t` (in addition to `==` and `/=`)
- Back to sorting ...
- The correct typing is:

```
quicksort :: Ord a => [a] -> [a]
```

Type classes

- If `t` is an instance of `Eq`, then `==` and `/=` are defined for `t`
- If `t` is an instance of `Ord`, then `<`, `<=`, `>`, `>=` are also defined for `t` (in addition to `==` and `/=`)
- Back to sorting ...
- The correct typing is:

```
quicksort :: Ord a => [a] -> [a]
```

- If `a` is an instance of `Ord`, `quicksort` is of type `[a] -> [a]`

Typing `elem`

- How can we type `elem`?

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

Typing `elem`

- How can we type `elem`?

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

- Consider the list?

```
funclist = [factorial, (+3), (*5)] :: [Int -> Int]
```

Typing `elem`

- How can we type `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`?

Equality

- Can we check `f == g` for functions?

Equality

- Can we check `f == g` for functions?
- `f x == g x` for all `x`?

Equality

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

Equality

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

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

Equality

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

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

- `factorial (-1)` does not terminate

Equality

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

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

- `factorial (-1)` does not terminate
- f == g implies that for all `x`, `f x` terminates iff `g x` does

- Can we write a function

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

such that `halting f x` is **`True`** iff `f x` terminates?

Equality

- Can we write a function

```haskell
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

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

The type class `Eq`

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

The type class `Eq`

- `Eq a` holds if `==`, `/=` are defined on `a`
- The typing for `elem` is:

```
elem :: Eq a => a -> [a] -> Bool
```

The type class `Eq`

- `Eq a` holds if `==`, `/=` are defined on `a`
- The typing for `elem` is:

```
elem :: Eq a => a -> [a] -> Bool
```

- If `Eq a` and `Eq b`, then `Eq (a,b)`, `Eq [a]`, `Eq [[a]]`, …

The type class `Eq`

- `Eq a` holds if `==`, `/=` are defined on `a`
- The typing for `elem` is:

```
elem :: Eq a => a -> [a] -> Bool
```

- If `Eq a` and `Eq b`, then `Eq (a,b)`, `Eq [a]`, `Eq [[a]]`, ...
- But we cannot extend `Eq a`, `Eq b` to `Eq (a -> b)`

The type class `Ord`

- If `Ord a` holds then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `a`

The type class `Ord`

- If `Ord a` holds then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `a`
- If `Ord a` then `Ord [a]` – lexicographic (dictionary) order

The type class `Ord`

- If `Ord a` holds then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `a`
- If `Ord a` then `Ord [a]` – lexicographic (dictionary) order
- If `Ord a` and `Ord b`, then `Ord (a,b)`

The type class `Ord`

- If `Ord a` holds then `<`, `<=`, `>`, `>=`, `==`, `/=` are defined for `a`
- If `Ord a` then `Ord [a]` – lexicographic (dictionary) order
- If `Ord a` and `Ord b`, then `Ord (a,b)`
- Cannot extend `Ord a`, `Ord b` to `Ord (a -> b)`

The type class Num

- Recall the function `sum`

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

The type class `Num`

- Recall the function `sum`

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

- `sum` requires `+` to be defined on list elements

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, and supports basic arithmetic operations

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, and supports basic arithmetic operations
- The correct typing for `sum`

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

Some other type classes

- `Integral`, `Frac` – subclasses of `Num`

Some other type classes

- `Integral`, `Frac` – subclasses of `Num`
- `Show` – values that can be displayed

Some other type classes

- `Integral`, `Frac` – subclasses of `Num`
- `Show` – values that can be displayed
- For a type `t` to be an instance of `Show`, we need a definition for the following function:

```
show :: a -> String
```

Some other type classes

- `Integral`, `Frac` – subclasses of `Num`
- `Show` – values that can be displayed
- For a type `t` to be an instance of `Show`, we need a definition for the following function:

```
show :: a -> String
```

- Provides a printable representation for values of type `a`

Some other type classes

- `Integral`, `Frac` – subclasses of `Num`
- `Show` – values that can be displayed
- For a type `t` to be an instance of `Show`, we need a definition for the following function:

```
show :: a -> String
```

- Provides a printable representation for values of type `a`
- The built-in datatypes are all instances of the expected type classes