Paradigmas de Programação

Week 6
Type Classes[1]

Alexandra Mendes

[1] Parts of these slides were taken from the slides kindly provided to me by Prof. Henrik Nilsson from University of Nottingham (UK) – look him up!

Introduction: what we saw previously

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables. Example:

```
length :: [a] -> Int
```

This means that type variables can be instanciated to different types:

```
Prelude> length ['a','b']
2
Prelude> length [1,2,3,4]
4
```

Introduction: what we saw previously

Overloaded Functions and Types

A polymorphic function is called overloaded if its type contains one or more class constraints. Example:

```
(+) :: Num a => a -> a -> a
```

A type that contains one or more class constraints is called overloaded (**Overloaded Type**)

Introduction: what we saw previously

Overloaded Functions and Types

Constrained type variables can be instantiated to any types that satisfy the constraints.

```
Prelude> 1.0 + 2.0
3.0
Prelude> 'a' + 'b'
<interactive>:4:5:
    No instance for (Num Char) arising from a use of +
    In the expression: 'a' + 'b'
    In an equation for it: it = 'a' + 'b'
```

Introduction: what we saw previously

Some Type Classes

Haskell has a number of type classes, including:

- Num - Numeric types. E.g:

```
(+) :: Num a => a -> a -> a
```

- Eq - Equality types. E.g:

```
(==) :: Eq a => a -> a -> Bool
```

- Ord - Ordered types. E.g:

```
(<) :: Ord a => a -> a -> Bool
```

Types vs Type Classes

- **Types:** Sets of values;
- **Types Classes:** Sets of Types.

Polymorphism vs Overloading/Constrained

- **(Unconstrained/Parametric) Polymorphism:** Type variables can be of any type. E.g.

  ```
  (.) :: (b -> c) -> (a -> b) -> (a -> c)
  ```

- **Constrained Types:** Type variables can be of any type that satisfies some constraints. E.g.

  ```
  print :: Show a => a -> IO ()
  ```

  Useful for when we don't want to limit a function to concrete types (e.g. Int, Bool) but we are unable to define the function without assuming some properties about the types.

Type Classes

- Type classes is one of the distinguishing features of Haskell
- Introduced to make ad hoc polymorphism, or overloading, less ad hoc
- Promotes reuse, making code more readable
- Central to elimination of all kinds of “boiler-plate” code[2] and sophisticated datatype-generic programming.
- Key reason why many practitioners like Haskell: lots of programming can happen automatically!

---

[2] Code that has to be included in many places with little or no alteration

Type Classes

- **What is the type of (==)?**

  E.g. the following both work:

  ```
  1 == 2
  'a' == 'b
  ```

  I.e., (==) can be used to compare both numbers and characters.

Type Classes

- **What is the type of (==)?**

  E.g. the following both work:

  `1 == 2`

  `'a' == 'b`

  I.e., (==) can be used to compare both numbers and characters.

- `Maybe (==) :: a -> a -> Bool?`

Type Classes

- **What is the type of (==)?**

  E.g. the following both work:

  `1 == 2`

  `'a' == 'b`

  I.e., (==) can be used to compare both numbers and characters.

- `Maybe (==) :: a -> a -> Bool?`

- No!!! Cannot work uniformly for arbitrary types!

Type Classes

A function like the identity function

```
id :: a -> a
id x = x
```

is polymorphic precisely because it works uniformly for all types: there is no need to “inspect” the argument.

But this is not always the case: **to compare two “things” for equality, they have to be inspected, and an appropriate method of comparison needs to be used.**

Type Classes

Moreover, some types do not in general admit a decidable equality. E.g. functions (when their domain is infinite).

Similar remarks apply to many other types. E.g.:

- We may want to be able to add numbers of any kind.
- But to add properly, we must understand what we are adding.
- Not every type admits addition.

Type Classes

Idea:

- Introduce the notion of a **type class**: a set of types that support certain related operations.
- **Constrain** those operations to **only** work for types belonging to the corresponding class.
- Allow a type to be **made an instance of** (added to) a type class by providing **type-specific implementations** of the operations of the class.

The Type Eq

```
class Eq a where
(==) :: a -> a -> Bool
```

$(==)$ is not a function, but a method of the type class Eq.

Its type signature is:

```
(==) :: Eq a => a -> a -> Bool
```

Eq a is a class constraint. It says that the equality method works for any type belonging to the type class Eq.

Instances of `Eq`

Various types can be made instances of a type class like **Eq** by providing implementations of the class methods for the type in question:

```
instance Eq Int where
  x == y = primEqInt x y

instance Eq Char where
  x == y = primEqChar x y
```

Instances of `Eq`

Suppose we have a data type:

```
data Answer = Yes | No | Unknown
```

How can we make `Answer` an instance of `Eq`?

Instances of `Eq`

Suppose we have a data type:

```
data Answer = Yes | No | Unknown
```

How can we make `Answer` an instance of `Eq`?

```
instance Eq Answer where
  Yes     == Yes     = True
  No      == No      = True
  Unknown == Unknown = True
  _       == _       = False
```

Instances of `Eq`

Consider the data type:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

Can `Tree` be made an instance of `Eq`?

Instances of `Eq`

Consider the data type:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

Can `Tree` be made an instance of `Eq`?

Yes, for any type a that is already an instance of Eq:

```
instance (Eq a) => Eq (Tree a) where
  Leaf a1      == Leaf a2      = a1 == a2
  Node t1l t1r == Node t2l t2r = t1l == t2l && t1r == t2r
  _            == _            = False
```

Instances of `Eq`

```
instance (Eq a) => Eq (Tree a) where
  Leaf a1       == Leaf a2       = a1 == a2
  Node t1l t1r == Node t2l t2r = t1l == t2l && t1r == t2r
  _             == _             = False
```

Note:

- (==) is used at type a when comparing a1 and a2;
- the use of (==) for comparing subtrees is a recursive call.

Deriving Type Classes

Instance declarations are often obvious and mechanical. Thus, for certain built-in classes (notably Eq, Ord , Show), Haskell provides a way to automatically derive instances, as long as:

- the data type is sufficiently simple
- we are happy with the standard definitions

Thus, we can do:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
            deriving Eq
```

Type Class Example

Type classes are similar to interfaces in Java in the sense that they define functions that have to be implemented by any data type that wants to be an instance of the class.

Example:

```
class Listable a where
  toList :: a -> [Int]
```

The class of things which can be converted to a list of Ints.

The type of `toList`:

```
toList :: Listable a => a -> [Int]
```

Type Class Example

```
class Listable a where
  toList :: a -> [Int]
```

Example of its use:

```
instance Listable Bool where
  toList True  = [1]
  toList False = [0]
```

Type Class Example

```
class Listable a where
  toList :: a -> [Int]
```

Example of its use:

```
instance Listable Bool where
  toList True  = [1]
  toList False = [0]
```

Exercise

Define `Int` as an instance of `Listable`.

Deriving Type Classes

- For user-defined typeclasses, GHC has no idea how their functions should be implemented when you try to associate the typeclass with a data type;
- For some typeclasses such as Show, Eq, and Ord where the functionality is simple enough, GHC can generate default implementations that you can overwrite.

From the Haskell 98 report:

The only classes in the Prelude for which derived instances are allowed are Eq, Ord, Enum, Bounded, Show, and Read (...)

You can read more details about this in `https://www.haskell.org/onlinereport/derived.html#derived-appendix`.

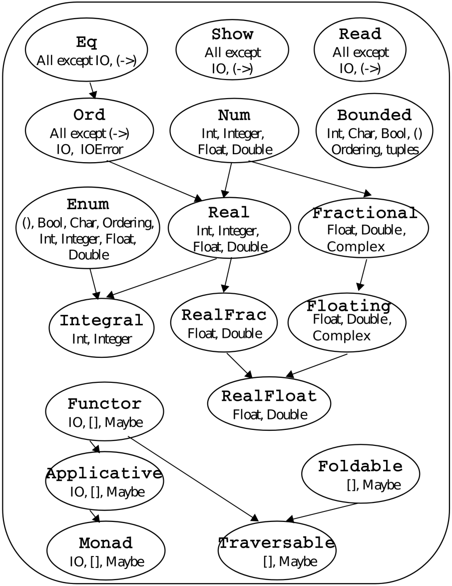
Class Hierarchy

Type classes form a hierarchy. E.g.:

```
class Eq a => Ord a where
(<=) :: a -> a -> Bool
...
```

`Eq` is a **superclass** of `Ord` ; i.e., any type in `Ord` must also be in `Eq`.

Type Classes



Some Basic Haskell Classes

```
class Eq a where
  (==), (/ =) :: a -> a -> Bool

class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max , min :: a -> a -> a

class Show a where
  show :: a -> String
```

Some Basic Haskell Classes

```
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a

class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
```

Type Classes: Example

```
data Maybe a = Nothing | Just a
               deriving Show
```

Uses the default definition of `show`.

If we want our own definition we should instead implement our own instance of `Show`:

```
data Maybe a = Nothing | Just a

instance (Show a) => Show (Maybe a) where
  show (Nothing) = show "Nothing␣Much"
  show (Just a) = show ("Just␣")++ show a ++("!")
```

Checking a Type Classes

If you want to check the definition of a class you can use the `:i` command in GHCi:

```
> :i Fractional

class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
   -- Defined in GHC.Real
instance Fractional Float -- Defined in GHC.Float
instance Fractional Double -- Defined in GHC.Float
```

Functor Type Class

- Previously, we saw the idea of mapping a function over each element of a list with the function `map`:

```
map :: (a -> b) -> [a] -> [b]
```

- However, mapping over each element of a data structure is not specific to lists and can be abstracted over a wide range of parameterised types;
- The class of types that support such a mapping function are called functors.

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b
```

Functor Type Class

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b
```

- For a parameterised type `f` to be an instance of the class `Functor`, it must support a function `fmap` of the specified type.

Functor Type Class

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b
```

- For a parameterised type `f` to be an instance of the class `Functor`, it must support a function `fmap` of the specified type.
- Intuition: `fmap` takes a function of type **a** -> **b** and a structure of type **f a** whose elements have type **a**, and applies the function to each element transforming it into a structure of type **f b** whose elements are now of type **b**.
- A **Functor** represents a type that can be mapped over.

Functor Type Class

Lists are instances of `Functor`:

```
instance Functor [] where
        fmap = map
```

We don't write

```
instance Functor [a] where ...
```

because from

```
fmap :: (a -> b) -> f a -> f b
```

we see that the `f` has to be a type constructor that takes one type. [a] is already a concrete type, while [ ] is a type constructor that takes one type and can produce types such as [Int].

Exercise

Consider the type:

```
data Tree a = Leaf a
            | Node a (Tree a) (Tree a)
```

Make the type `Tree Int` an instance of the class `Listable`.
You will need to add to the top of your script the following line:

```
{-# LANGUAGE FlexibleInstances #-}
```

This is because the Haskell specification requires type variables in the declaration of instances of types such as `Tree a` which is polymorphic.

Exercise

Using the GHCi, list what are the superclasses of `Integral` and what are its instances.