BAHUG 101 - Lecture 4

21th October 2015

Outline of Today’s Lecture

- Typeclasses
- Monoids
- Functors

Typeclasses

Types of Polymorphism

There are two types of polymorphisms in Haskell.

Types of Polymorphism

There are two types of polymorphisms in Haskell.

- Parametric Polymorphism: The one we know and love. Allows for functions that work over *any* type.

```
length :: [a] → Int
```

Types of Polymorphism

There are two types of polymorphisms in Haskell.

- Parametric Polymorphism: The one we know and love. Allows for functions that work over *any* type.

```
length :: [a] → Int
```

- Ad-hoc Polymorphism: functions that work on a set of types, but not all types. For example, `(+)`.

```
(+) :: SomethingAddable → SomethingAddable → SomethingAddable
```

Types of Polymorphism

There are two types of polymorphisms in Haskell.

- Parametric Polymorphism: The one we know and love. Allows for functions that work over *any* type.

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

- Ad-hoc Polymorphism: functions that work on a set of types, but not all types. For example, `(+)`.

```haskell
(+) :: SomethingAddable -> SomethingAddable -> SomethingAddable
```

As an example, the following can be used with `(+)`.

- `Int`
- `Float`, `Double`

And the following are not

- `Char`, `String`

Haskell Type Classes

Haskell allows us to perform ad-hoc polymorphism through *type classes*.

- Type Class: A set of types and set of operations on that apply to each type.

Haskell Type Classes

Haskell allows us to perform ad-hoc polymorphism through *type classes*.

- Type Class: A set of types and set of operations on that apply to each type.

As an example, here is a common Haskell type class.

```haskell
class Eq a where
  (==) :: a → a → Bool
  (≠) :: a → a → Bool
```

Note that **a** here is a type parameter.

Haskell Type Classes

Haskell allows us to perform ad-hoc polymorphism through *type classes*.

- Type Class: A set of types and set of operations on that apply to each type.

As an example, here is a common Haskell type class.

```
class Eq a where
  (==) :: a → a → Bool
  (≠) :: a → a → Bool
```

Note that `a` here is a type parameter.

Common types that are in the `Eq` type class are `Int`, `Float`, `Char`, etc.

Writing a function using a type class

Let’s look a the `(==)` function.

```haskell
(==) :: Eq a ⇒ a → a → Bool
```

Writing a function using a type class

Let's look a the `(==)` function.

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

The `Eq a` above is a _type class constraint'.

We can only use `(==)` on types that implement the `Eq` type class.

Writing a function using a type class

Let’s look a the `(==)` function.

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

The `Eq a` above is a _type class constraint‘.

We can only use `(==)` on types that implement the `Eq` type class.

The compiler figures out which `(==)`, etc function to use at compile time.

Example implementing Eq

Here we will take some simple Foo type.

```
data Foo = F Int | G Char
```

Example implementing **Eq**

Here we will take some simple `Foo` type.

```haskell
data Foo = F Int | G Char
```

and make it an instance of `Eq`

```haskell
instance Eq Foo where
  (F i1) == (F i2) = i1 == i2
  (G c1) == (G c2) = c1 == c2
  _ == _ = False
```

Example implementing **Eq**

Here we will take some simple `Foo` type.

```
data Foo = F Int | G Char
```

and make it an instance of `Eq`

```
instance Eq Foo where
  (F i1) == (F i2) = i1 == i2
  (G c1) == (G c2) = c1 == c2
  _ == _ = False

  foo1 /= foo2 = not (foo1 == foo2)
```

Default implementations in type classes

We could have also have defined a defaumt implementation.

```
class Eq a where
  (==) :: a → a → Bool
  (≠) :: a → a → Bool
  x ≠ y = not (x == y)
```

Default implementations in type classes

We could have also have defined a defaumt implementation.

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

and then just done

```haskell
instance Eq Foo where
  (F i1) == (F i2) = i1 == i2
  (G c1) == (G c2) = c1 == c2
  _ == _ = False
```

Default implementations in type classes

We could have also have defined a defaumt implementation.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

and then just done

```
instance Eq Foo where
  (F i1) == (F i2) = i1 == i2
  (G c1) == (G c2) = c1 == c2
  _ == _ = False
```

We can also override a default.

How **Eq** is actually declared

The **Eq** class is actually declared like this:

```
class Eq a where
  (==), (≠) :: a → a → Bool
  x == y = not (x ≠ y)
  x ≠ y = not (x == y)
```

How **Eq** is actually declared

The **Eq** class is actually declared like this:

```
class Eq a where
  (==), (≠) :: a → a → Bool
  x == y = not (x ≠ y)
  x ≠ y = not (x == y)
```

For **Eq**, we can define *either* **(==)** *or* **(≠)**, or both.

Automatic type class instances

We can automatically derive certain type class instances like so.

```
data Foo’ = F’ Int | G’ Char
  deriving (Eq, Ord, Show)
```

Automatic type class instances

We can automatically derive certain type class instances like so.

```
data Foo' = F' Int | G' Char
  deriving (Eq, Ord, Show)
```

The following type classes can have instances automatically derived.

- `Eq`: Things that can be tested for equality.
- `Ord`: Things that can be tested for order.
- `Enum`: Things that can be enumerated.
- `Ix`: Things that can be mapped onto the integers.
- `Bounded`: Things that have a lower and upper bound.
- `Show`: Things that can be turned into `String`.
- `Read`: Things that can be turned from a `String` into a data type (`read :: Read a => String -> a`).

Type classes and Java interfaces

Type classes in Haskell are similar to Java interfaces, but there are some differences.

1. Type classes often come with a set of mathematical laws that *should* be followed by all instances (ex, `Num`).

Type classes and Java interfaces

Type classes in Haskell are similar to Java interfaces, but there are some differences.

1. Type classes often come with a set of mathematical laws that *should* be followed by all instances (ex, `Num`).

2. Type class instances are declared separately from the declaration of the corresponding types.

Type classes and Java interfaces

Type classes in Haskell are similar to Java interfaces, but there are some differences.

1. Type classes often come with a set of mathematical laws that *should* be followed by all instances (ex, `Num`).

2. Type class instances are declared separately from the declaration of the corresponding types.

3. The types that can be specified for type class methods are more general and flexible.

Type classes and Java interfaces

Type classes in Haskell are similar to Java interfaces, but there are some differences.

1. Type classes often come with a set of mathematical laws that *should* be followed by all instances (ex, `Num`).
2. Type class instances are declared separately from the declaration of the corresponding types.
3. The types that can be specified for type class methods are more general and flexible.
4. Haskell type classes can also easily handle binary (or ternary, or ...) methods, as in `Num`.

Multi-parameter type classes

We can have type classes with multiple type parameters using `MultiParamTypeClasses`.

```
{-# LANGUAGE MultiParamTypeClasses #-}

class Blerg a b where
  blerg :: a → b → Bool
```

`blerg` is similar to multiple dispatch.

Functional Dependencies

We can also use *functional dependencies*.

```haskell
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleInstances #-}

class Extract a b | a -> b where
  extract :: a -> b
```

Here, `a` uniquely determines `b`.

Functional Dependencies

We can also use *functional dependencies*.

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

class Extract a b | a -> b where
  extract :: a -> b
```

Here, `a` uniquely determines `b`.

As an example

```
instance Extract (a, b) a where
  extract (x, y) = x
```

Functional Dependencies

We can also use *functional dependencies*.

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

class Extract a b | a -> b where
  extract :: a -> b
```

Here, `a` uniquely determines `b`.

As an example

```
instance Extract (a, b) a where
  extract (x, y) = x
```

However, because of the functional dependency, we cannot create the instance:

```
instance Extract (a, b) b where ...
```

because the type `(a,b)` uniquely determines `a`.

Monoids

Monoid definition

For a type `m` and an operation `(<>) :: m -> m -> m`, we have a *monoid* when

Monoid definition

For a type `m` and an operation `(<>) :: m -> m -> m`, we have a *monoid* when

1. there exists a particular element `mempty :: m` such that

- `x <> mempty == x`, and

Monoid definition

For a type `m` and an operation `(<>) :: m -> m -> m`, we have a *monoid* when

1. there exists a particular element `mempty :: m` such that

- `x <> mempty == x`, and
- `mempty <> x == x` ; and

Monoid definition

For a type `m` and an operation `(<>) :: m -> m -> m`, we have a *monoid* when

1. there exists a particular element `mempty :: m` such that
   - `x <> mempty == x`, and
   - `mempty <> x == x` ; and
2. the operation `(<>)` is associative.

`(a <> b) <> c == a <> (b <> c)`.

Monoid Haskell Type Class

```
class Monoid m where
  mempty  :: m
  mappend :: m → m → m

  mconcat :: [m] → m      — this can be omitted from Monoid instan
  mconcat = foldr mappend mempty

(<>) :: Monoid m ⇒ m → m → m     — infix operator for convenienc
(<>) = mappend
```

Example Monoids

There are a great many **`Monoid`** instances available. Perhaps the easiest one is for lists:

```haskell
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Monoids are useful whenever an operation has to combine results.

Example Monoid function

Example function that finds numbers below 100 that are divisible by 5 or 7, and accumuates them

```
intInts :: Monoid m ⇒ (Integer → m) → m    — interesting ints!
intInts mk_m = go [1..100]    — [1..100] is the list of numbers fro
  where go [] = mempty
        go (n:ns)
          | let div_by_5 = n ‘mod‘ 5 == 0
                div_by_7 = n ‘mod‘ 7 == 0
          , (div_by_5 || div_by_7) && (not (div_by_5 && div_by_7))
          = mk_m n <> go ns
          | otherwise
          = go ns
```

The `mk_m` parameter converts an `Integer` into whatever monoid the caller wants.

Gettings results out

Here, we can get these results as a list:

```
intIntsList :: [Integer]
intIntsList = intInts (:[])
```

Gettings results out

Here, we can get these results as a list:

```haskell
intIntsList :: [Integer]
intIntsList = intInts (:[])
```

Suppose we want to combine the numbers as a product, instead of as a list. You might be tempted to say

```haskell
intIntsProduct :: Integer
intIntsProduct = intInts id
```

Gettings results out

But this will work

```
data Product a = Product a
instance Num a ⇒ Monoid (Product a) where
  mempty                          = Product 1
  mappend (Product x) (Product y) = Product (x * y)

getProduct :: Product a → a
getProduct (Product x) = x
```

Gettings results out

But this will work

```haskell
data Product a = Product a
instance Num a => Monoid (Product a) where
  mempty                            = Product 1
  mappend (Product x) (Product y) = Product (x * y)

getProduct :: Product a -> a
getProduct (Product x) = x

intIntsProduct :: Integer
intIntsProduct = getProduct $ intInts Product
```

Functor

Definition of a functor

There is one last type class you should learn about, **`Functor`**:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

It may be helpful to see some instances before we pick the definition apart:

```
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```