Haskell

A functional paradigm exemplar and other aspects (cont'd)

Ana Oprescu

UvA

February 13, 2020

Typeclasses

Types can be part of typeclasses (members)[1]:

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

- Eq: type supports == en /=
- Ord: type is ordered
- Show: type can be represented as a string
- Read: string can be represented as the type
- Enum: type is sequentially ordered
- Bounded: type has lower- and upper-bounds
- Num: type acts as a number
- Integral: type is an integer number (Int/Integer)
- Floating: type is a floating point number (Float/Double)

[1] https://www.haskell.org/tutorial/classes.html

The Eq typeclass

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

```
*MyEq> [1,2] MyEq.== [3,4]

<interactive>:52:1: error:
     No instance for (MyEq.Eq [Integer])
      arising from a use of  MyEq .==
      In the expression: [1, 2] MyEq.== [3, 4]
    In an equation for   it   : it = [1, 2] MyEq.== [3, 4]
```

Type system

- Type checker:

  static → at compile-time strong → No unchecked runtime type error

- Type inference: algorithmic reasoning about types of variables and parameters

```
Prelude> [1,'a']
<interactive>:62:2: error:
       No instance for (Num Char) arising from the literal    1
        In the expression: 1
      In the expression: [1, 'a']
      In an equation for    it    : it = [1, 'a']
```

- Type classes: introduce overloading as a principle → ad-hoc polymorphism [2]

  class extension

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

  - multiple inheritance

    ```
    class  (Eq a, Show a) => ShowOrd a  where  ...
    ```

[2] https://www.researchgate.net/publication/2710954_How_to_Make_Ad-Hoc_Polymorphism_Less_Ad_Hoc
    ```

Type Synonyms

We can define aliases using the **type** keyword:

Lecture.hs

```haskell
type Feedback = (Int, Int)
```

Newtypes

We can define new types using the **newtype** keyword [3]. These will not have the same typeclasses inherited, but the same representation.

```
module MN where
newtype Natural = MakeNatural Integer
toNatural                :: Integer -> Natural
toNatural x | x < 0      = error "Can't create negative naturals!"
            | otherwise = MakeNatural x

fromNatural              :: Natural -> Integer
fromNatural (MakeNatural i) = i
instance Num Natural where
  fromInteger          = toNatural
  x + y                = toNatural (fromNatural x Prelude.+ fromNatural y)
  x - y                = let r = fromNatural x Prelude.- fromNatural y in
                             if r < 0 then error "Unnatural subtraction"
                                      else toNatural r

  x * y                = toNatural (fromNatural x Prelude.* fromNatural y)
  abs x                = x
  signum x             = toNatural (Prelude.signum $ fromNatural x)
```

3

Custom datatypes

We can define our own types using the **data** keyword:

Lecture.hs

```haskell
data Color = Red | Yellow | Blue | Green | Orange | Purple
             deriving (Eq, Show, Bounded, Enum)
```

```
Prelude> [minBound..maxBound] :: [Color]
[Red,Yellow,Blue,Green,Orange,Purple]
Prelude> let foo = Red
Prelude> foo
Red
Prelude> :t foo
foo :: Color
```

Playing with Types[4]

- "casting"

```
Prelude> let a = 3 :: Int
Prelude> let b = 4 :: Integer
Prelude> a+b

<interactive>:11:3: error:
        Couldn't match expected type    Int     with actual type     Integer
        In the second argument of    (+)    , namely    b
      In the expression: a + b
      In an equation for    it    : it = a + b
Prelude> a + read (show b)::Int
7
```

[4] http://learnyouahaskell.com/types-and-typeclasses#type-variables

Playing with Types

- "inferring"

```
Prelude> elem' a = foldr (==a) . (||)
    <interactive>:87:18: error:
     Couldn't match type    Bool    with    (Bool -> Bool) -> Bool -> Bool
     Expected type: a -> (Bool -> Bool) -> Bool -> Bool
     Actual type: a -> Bool
      In the first argument of    foldr    , namely    (== a)
      In the first argument of    (.)    , namely    foldr    (== a)
      In the expression: foldr (== a) . (||)
```

```
Prelude> elem' a = foldr (==a) . (||) False
<interactive>:88:18: error:
    Couldn't match type    Bool    with    Bool    -> Bool
Expected type: a -> Bool -> Bool
Actual type: a -> Bool
    In the first argument of    foldr    , namely    (== a)
  In the first argument of    (.)    , namely    foldr    (== a)
  In the expression: foldr (== a) . (||) False
```

Playing with Types

- "inferring" cont'd

```
Prelude> elem' a = foldr ((==a) . (||)) False
<interactive>:89:18: error:
    Couldn't match type   Bool   with   Bool   -> Bool
Expected type: Bool -> Bool -> Bool
Actual type: Bool -> Bool
    Possible cause:   (.)   is applied to too many arguments
In the first argument of   foldr   , namely   ((== a) . (||))
In the expression: foldr ((== a) . (||)) False
In an equation for   elem '   : elem' a = foldr ((== a) . (||)) False
```

Playing with Types

- "inferring" cont'd

```
Prelude> elem' a = foldr ((==a) . (||)) False
<interactive>:89:18: error:
    Couldn't match type   Bool   with   Bool   -> Bool
Expected type: Bool -> Bool -> Bool
Actual type: Bool -> Bool
    Possible cause:   (.)   is applied to too many arguments
In the first argument of   foldr   , namely   ((== a) . (||))
In the expression: foldr ((== a) . (||)) False
In an equation for   elem '   : elem' a = foldr ((== a) . (||)) False
```

```
Prelude> elem' a = foldr ((||) . (==a)) False
```

Type and data constructors

Tree is a type constructor, while Leaf and Node are data constructors:

Lecture.hs

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

Using Type constructors and functors

You make the Tree datatype a functor by:

```
instance Functor Tree where
  fmap f (Leaf x)       = Leaf   (f x)
  fmap f (Node t1 t2) = Node (fmap f t1) (fmap f t2)
```

Functors are also known as <$>

Monads

For a gentle introduction to Functors, Applicatives and Monads go to http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

```
class Monad m where
  (>>=)  :: m a -> (  a -> m b) -> m b -- the bind operator
  (>>)   :: m a ->  m b         -> m b -- sequence, but discard intermediate
  return ::   a                 -> m a -- wrap it back for purity
  fail   :: String -> m a
```

The Maybe Monad

```
Maybe a = Nothing | Just a
instance Monad Maybe where
  Nothing >>= _ = Nothing
  (Just x) >>= f = f x
```

```
index' :: Integer -> [a] -> Maybe a
index' i xs = fst . foldl findKey (Nothing, 0) $ xs
    where findKey (value, j) k
            | i == j    = (Just k, j + 1)
                | otherwise = (value, j + 1)

mapM f [] = return []
mapM f (x:xs) = do y <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

Let's revisit Natural

```
module MN where ...

instance Show Natural where
  show x             = show (fromNatural x)
```

```
*Main> :l maybeitem.hs makenatural.hs
[1 of 2] Compiling MN               ( makenatural.hs, interpreted )
[2 of 2] Compiling Main             ( maybeitem.hs, interpreted )
Ok, two modules loaded.
*Main> index' 3 [MN.MakeNatural 3, MN.MakeNatural 4]
Nothing
*Main> :t (index' 3 [MN.MakeNatural 3, MN.MakeNatural 4])
(index' 3 [MN.MakeNatural 3, MN.MakeNatural 4]) :: Maybe MN.Natural
```