Functional Programming
Lecture 9

Rostislav Horčík
Niklas Heim

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz
heimnikl@fel.cvut.cz

Pattern matching on records

Pattern matching on records

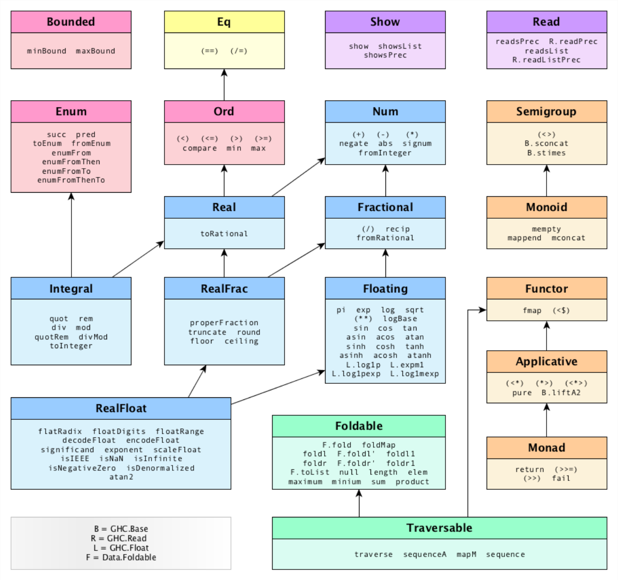
```haskell
data Vector a = Vec { x::a, y::a, z::a }
                deriving Show

isZero :: (Eq a, Num a) => Vector a -> Bool
isZero Vec{x=0,y=0,z=0} = True
isZero _ = False

last :: Vector a -> a
last Vec{z=w} = w
```

With the extension *{-# LANGUAGE RecordWildCards #-}*

```haskell
norm :: Floating a => Vector a -> a
norm Vec{..} = sqrt (x^2 + y^2 + z^2)
```

Type classes

Zoo of typeclasses



Read

`Read` is a type class opposite to `Show`. It allows to parse strings into values for all instances of `Read` via the function

```haskell
read :: Read a => String -> a
```

`read` is polymorphic but sometimes we need an explicit type annotation.

```haskell
> read "3" -- fails

> read "3" :: Int
3

> read "[1,2,3]" :: [Float]
[1.0,2.0,3.0]
```

Type classes of parametric types

Higher-order functions

Familiar higher-order functions are available in Haskell too.

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
foldl ::
  Foldable t => (b -> a -> b) -> b -> t a -> b
foldr ::
  Foldable t => (a -> b -> b) -> b -> t a -> b
```

```
> foldl (+) 0 [1,2,3]
6
```

Functor

```haskell
map :: (a -> b) -> [a] -> [b]
mapMap :: (a -> b) -> Map k a -> Map k b
treeMap :: (a -> b) -> Tree a -> Tree b
```

Functor is a type class collecting type constructors that create structure we can map over.

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

<$> is an infix operator equivalent to `fmap`

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

Functor example

```haskell
data Tree a = Tree a [Tree a] deriving Show

tree :: Tree Int
tree = Tree 1 [Tree 2 [Tree 3 []], Tree 4 []]

instance Functor Tree where
  fmap f (Tree x []) = Tree (f x) []
  fmap f (Tree x ts) = Tree (f x)
                            (map (fmap f) ts)
```

Kinds

Kinds are “types” of types and type constructors

`*`	A specific type like `Int` or `Int -> Char`
`* -> *`	A type constructor that given a type creates a type, e.g. `Maybe`
`* -> * -> *`	A type constructor that given two types creates a type, e.g. `Either`
`* -> Constraint`	A constructor of a type constraint e.g. `Num`

```
data Either a b = Left a | Right b
```

GHCi command to display kinds is `:k`.

Handling errors in pure code

Failing computations

To define safe operations in Haskell, we can use

```haskell
data Maybe a = Nothing | Just a
```

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Just (head xs)
```

```haskell
safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs
```

Mapping over Maybe

```haskell
add1ToHead :: [Int] -> Int
add1ToHead = (+1) . head
```

The following fails because `(+1)` expects a numeric type not `Maybe Int`.

```haskell
add1ToHead :: [Int] -> Maybe Int
add1ToHead = (+1) . safeHead
```

Possible solution that does not scale:

```haskell
add1Maybe :: Maybe Int -> Maybe Int
add1Maybe Nothing = Nothing
add1Maybe (Just n) = Just (n + 1)
```

Maybe as Functor

Instead we need a universal lifting of `a -> b` to `Maybe a -> Maybe b`.

```
lift :: (a -> b) -> Maybe a -> Maybe b
```

But this is just `fmap` from `Functor`.

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

```
safeAdd1ToHead :: [Int] -> Maybe Int
safeAdd1ToHead = fmap (+1) . safeHead
```

Composing failing computations

```haskell
second :: [a] -> a
second = head . tail
```

This fails because `safeHead` expects `[a]` not `Maybe [a]`.

```haskell
safeSecond :: [a] -> Maybe a
safeSecond = safeHead . safeTail
```

This does not help either as the resulting type is
`Maybe (Maybe a)`.

```haskell
safeSecond :: [a] -> Maybe a
safeSecond = (fmap safeHead) . safeTail
```

Composing failing computations

```
safeSecond :: [a] -> Maybe a
safeSecond xs =
  let xs' = safeTail xs
  in case xs' of
       Nothing -> Nothing
       Just xs'' -> safeHead xs''
```

This approach does not scale well.

safeFourth

```haskell
safeFourth :: [a] -> Maybe a
safeFourth xs =
  let xs' = safeTail xs
  in case xs' of
       Nothing -> Nothing
       Just xs1 ->
         let xs1' = safeTail xs1
         in case xs1' of
              Nothing -> Nothing
              Just xs2 ->
                let xs2' = safeTail xs2
                in case xs2' of
                     Nothing -> Nothing
                     Just xs3 -> safeHead xs3
```

Composing failing computations

```haskell
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing _ = Nothing
andThen (Just x) f = f x

safeSecond :: [a] -> Maybe a
safeSecond xs = safeTail xs `andThen` safeHead

safeFourth :: [a] -> Maybe a
safeFourth xs =
  safeTail xs `andThen`
  safeTail `andThen`
  safeTail `andThen`
  safeHead
```

Either

Error reporting is often done via

```
data Either a b = Left a | Right b
```

`Either` has two parameters so its kind is `* -> * -> *`.

```
safeDiv :: Int -> Int -> Either String Int
safeDiv _ 0 = Left "Division by 0 error"
safeDiv x y = Right (x `div` y)
```

Summary

- A type class defines an interface for types.
- `Functor` is a type class for mappable type constructors.
- `Maybe` represents failing computations.
- `Maybe` is an instance of `Functor`.
- Composing of failing computation can be done by a higher-order function of type
  `Maybe a -> (a -> Maybe b) -> Maybe b`.
- Error reporting is done via `Either`.

JSON example

```haskell
data JValue = JString String
            | JNumber Double
            | JBool Bool
            | JNull
            | JObject [(String, JValue)]
            | JArray [JValue]
              deriving (Eq, Show, Ord)
```

```haskell
JObject [
  ("id", JNumber 103),
  ("name", JString "John"),
  ("courses",
   JArray [JString "FUP", JString "ZUI"])
]
```

Type classes

Type classes allow us to implement ad hoc polymorphisms by overloading function names.

```
class JSON a where
    toJValue :: a -> JValue
```

```
instance JSON Double where
    toJValue = JNumber

instance JSON Bool where
    toJValue = JBool
```

But the following fails as `String=[Char]`:

```
instance JSON String where
    toJValue = JString
```

Type class instances

Type class instances can be defined only for basic data types or type constructors over type variables. To overcome that in GHC, we must compile our file with the pragma

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

```haskell
instance JSON String where
    toJValue = JString
```

But the following is an overlapping instance with the above instance as `String=[Char]`

```haskell
instance JSON a => JSON [a] where
    toJValue = JArray . map toJValue
```

This can be handled with pragmas `{-# OVERLAPPING #-}` and `{-# OVERLAPPABLE #-}`

Solution

To overcome this issue we can introduce a wrapper:

```
newtype Str = Str String deriving (Eq, Show, Ord)
```

`newtype` is like `data` with only single data constructor. Its implementation is more efficient.

Then we change

```
data JValue = JString Str
            | ...

instance JSON Str where
    toJValue = JString
```

Case expression

Conditional expression allowing to control the evaluation based on the value of an expression by pattern matching.

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

```
describeList :: [a] -> String
describeList xs = "The list is "
                    ++ case xs of
                         [] -> "empty."
                         [_] -> "a singleton list."
                         _ -> "a longer list."
```

Case expression

The function definition via equations

```
f p11 ... p1k = e1
...
f pn1 ... pnk = en
```

where each `pij` is a pattern, is semantically equivalent to:

```
f x1 x2 ... xk = case (x1, ... , xk) of
                   (p11, ..., p1k) -> e1
                   ...
                   (pn1, ..., pnk) -> en
```