Haskell - features
Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

Spring 2018

Outline

The Haskell type system

- Data types
- Type synonyms
- Type Classes

Features

- Functors
- Maps
- Monoids - Calculating
- Folding
- Monads - Side effects tamed

- The Haskell type system
  - Data types
  - Type synonyms
  - Type Classes
- Features
  - Functors
  - Maps
  - Monoids - Calculating
  - Folding
  - Monads - Side effects tamed

data types

Elm: Union type

```
data Bool = False | True
```

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

Custom data types

```haskell
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

```haskell
data Person = { name :: String
              , age :: Int
              , email :: String
              } deriving (Show)
```

type Synonyms

Elm: `type alias`

```
type Point = (Float, Float)
data Shape = Circle Point Float
           | Rectangle Point Point
```

```
type PhoneBook = [(String, String)]

phoneBook :: PhoneBook
phoneBook = [ ... ]
```

Build in Type Classes

- `Ord` ordered type, supporting `<`, `>`, ...
- `Eq` equatable type, supporting `==` and `/=`
- `Enum` enumeratable type, works with `[1..10]`
- `Bounded` types with `minBound` and `maxBound`
- `Num` numeral type including integer and real numbers
  - `Integral` integer numbers
  - `Floating` real numbers
- `Show` showable type, supporting `show` (*toString*) function
- `Read` showable type, supporting `read` (*fromString*) function

Deriving type classes

```haskell
data Person = { name :: String
              , age :: Int
              , email :: String
              } deriving (Show, Eq)
```

```haskell
kurt = Person { name = "Kurt"
              , age = 25
              , email = "k@mail.dk"
              }
sonja = Person "Sonja" 28 "sonja@post.dk"
```

```
ghci> kurt
Person {name = "Kurt", age = 25, email = "k@mail.dk"}

ghci> kurt == sonja
False
ghci> sonja == sonja
True
```

Defining type classes

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

Defining type classes

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

```haskell
data Semaphore = Red | Yellow | Green | OutOfOrder
                 deriving (Show)
```

```haskell
instance Eq Semaphore where
  Red == Red = True
  (==) Yellow Yellow = True
  Green == Green = True
  _ == _ = False
```

Using defined type class

```
hgci> Red == Red
True
hgci> Red == Yellow
False
hgci> OutOfOrder == OutOfOrder
False
```

Exercise 1 - YesNo type class

Create a type class "`YesNo`" that defines the functions `true a` and `false a` both returning a `Bool`

Define instances of YesNo for lists, integers, Maybe's and booleans:

List	Empty lists yields false other lists true
Integers	0 yields false other numbers true
Maybe	Nothing yields false other Maybe's true
Bool	false yields false true yields true

The Haskell type system

- Data types
- Type synonyms
- Type Classes

Features

- Functors
- Maps
- Monoids - Calculating
- Folding
- Monads - Side effects tamed

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

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

Remember `map` on Lists:

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

Lists are functors

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

cphbusiness COPENHAGEN BUSINESS ACADEMY

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

Remember map on Lists:

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

Lists are functors

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

Maybe is also a functor:

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

Change the `package.yaml` file, add `containers` to dependencies:

```
dependencies:
- base >= 4.7 && < 5
- containers
```

```
hgci> import qualified Data.Map as Map
hgci> ns = Map.fromList [(1,"One"),(2,"Two"),(3,"Three")]
hgci> Map.lookup 2 ns
Just "Two"
hgci> Map.lookup 7 ns
Nothing
hgci> ns2 = Map.insert 7 "Seven" ns
hgci> Map.lookup 7 ns2
Just "Seven"
```

Monoids

```haskell
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty

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

instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  mappend (Sum x) (Sum y) = Sum (x + y)
```

```
ghci> import Data.Monoid
ghci> [1, 2, 3] `mappend` [7, 9, 13]
[1, 2, 3, 7, 9, 13]
ghci> mconcat ["Hello", " ", "World", "!"]
"Hello World!"
ghci> mappend (Sum 7) (Sum 17)
Sum {getSum = 24}
ghci> mconcat [(Product 7), (Product 9), (Product 13)]
Product {getProduct = 819}
```

Product is defined as[1]:

```
newtype Product a = Product { getProduct :: a }
    deriving (Eq, Ord, Read, Show, Bounded)
```

Define it as an instance of Monoid

[1] newtype works as data with one constructor with one argument

Folding

```haskell
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  ...
  foldl :: (b -> a -> b) -> b -> t a -> b
  ...
```

```haskell
hgci> import Data.Monoid
hgci> foldMap Sum [1..100]
Sum {getSum = 5050}
hgci> foldr (\x acc -> x * 3 : acc) [] [1..5]
[3,6,9,12,15]
hgci> foldl (+) 0 [1..100]
5050
```

```
main = do
  line <- getLine
  if null line
    then return ()
    else do
      putStr $ map toUpper line
      putStrLn " in upper case"
      main
```

```
print = putStrLn . show
```

```
ghci> mapM print [7, 9, 13]
7
9
13
```

```
main = do
  ideas <- forM [7, 9, 13] (\i -> do
    putStrLn "What idea associates with"++ show i ++"?"
    getLine
    )
  putStr "The ideas was: "
  mapM putStrLn ideas
```