MOD08: Functional Programming

Advanced Typeclasses

Marco Gerards

10-05-2019

Slack

- Programming Paradigms has a Slack channel
  - but is not used often...
- Ask questions outside scheduled hours
- Help from TAs and each other
- Canvas (PP) ⇒ General Information ⇒ Questions and Help

Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Parser combinators and ParSec (application)
	- Advanced type classes
4	- Code generation (application)
5-7	- Project

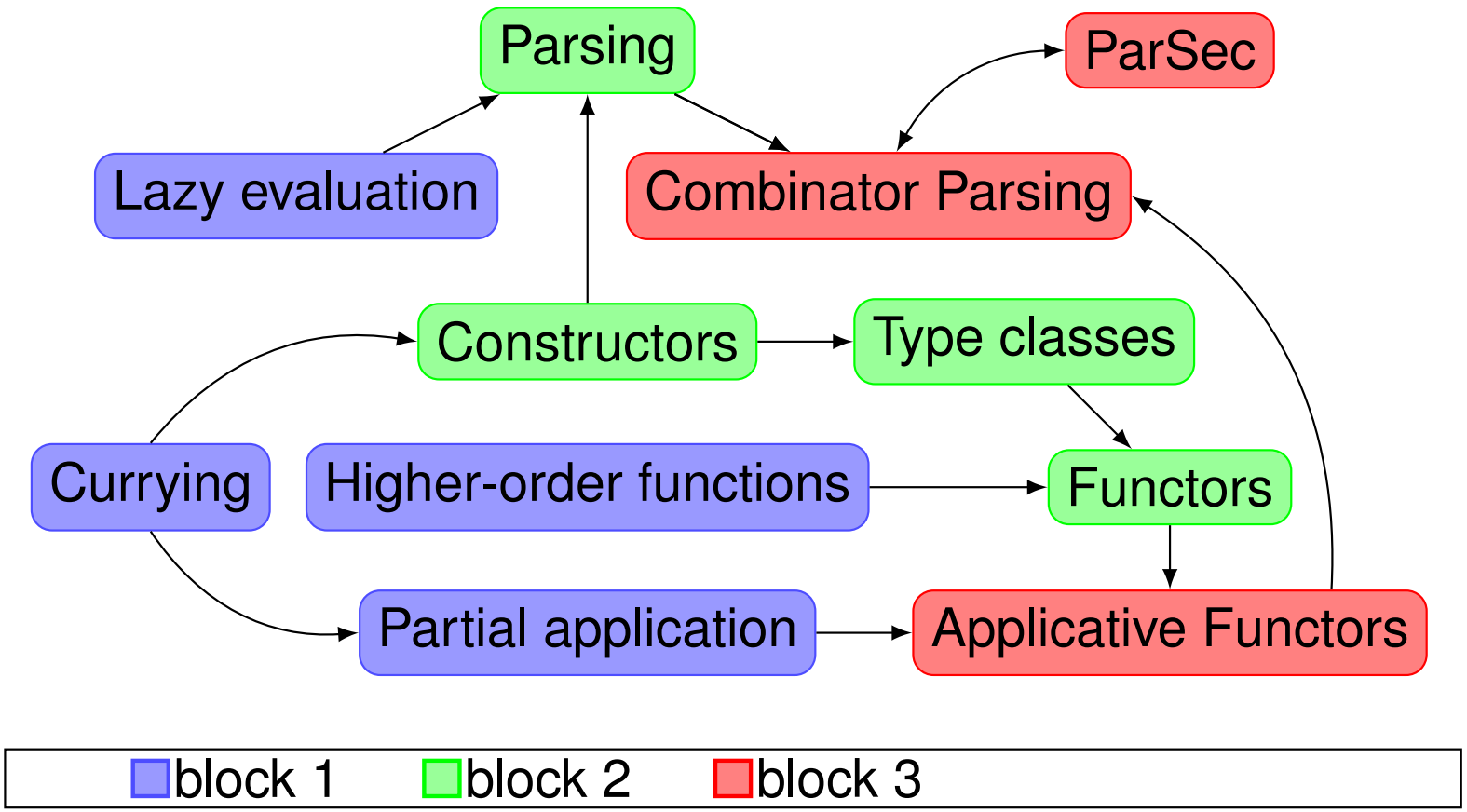
Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Parser combinators and ParSec (application)
	- Advanced type classes
4	- Code generation (application)
5-7	- Project

This lecture: learning goals

- Explain and use the typical types and data structures in FP
  - Applicative functors: extension of functors with currying
  - Alternative: Monoid on Applicative functors
  - Foldables: generalisation of `foldl` and `foldr`
  - Monoids: commutative binary operations with identity
    - `+` and `0`
    - `*` and `1`
    - `++` and `[]`
    - ...

Connection of some of the topics between blocks



Question 1

Consider the following type.

```
data Q = P {
  r :: String
}
```

What are the types of `P` and `r`?

A
```
r :: String
P :: String
```

B
```
r :: String -> Q
P :: String -> Q
```

C
```
r :: String
P :: String -> Q
```

D
```
r :: Q -> String
P :: String -> Q
```

Question 2

Consider the following Haskell expression

```
data F x = G (x -> x)
```

What is the kind of `F`?

A `*`

B `* -> *`

C `(* -> *) -> *`

D Unknown, since `x` is not yet given.

Applicative

Functors

```
x = Just 1
y = Just 2

z = (+2) <$> x
```

- Functors: defined for types that can be mapped over
- `fmap :: Functor f => (a -> b) -> f a -> f b`
- Maps a function `a -> b` “over” `f a`

Functors

```
x = Just 1
y = Just 2

z = (+2) <$> x
```

- Functors: defined for types that can be mapped over
- `fmap :: Functor f => (a -> b) -> f a -> f b`
- Maps a function `a -> b` "over" `f a`
- Problem: how about mapping curried functions?
  - How to add `x` and `y`?

Functors

```
x = Just 1
y = Just 2

z = (+2) <$> x
```

- Functors: defined for types that can be mapped over
- `fmap :: Functor f => (a -> b) -> f a -> f b`
- Maps a function `a -> b` "over" `f a`
- Problem: how about mapping curried functions?
  - How to add `x` and `y`?
- How about?

```
fmap2 :: Functor f => (a->b->c) -> f a -> f b -> f c
  ...
```

Functors

```
x = Just 1
y = Just 2

z = (+2) <$> x
```

- Functors: defined for types that can be mapped over
- `fmap :: Functor f => (a -> b) -> f a -> f b`
- Maps a function `a -> b` "over" `f a`
- Problem: how about mapping curried functions?
  - How to add `x` and `y`?
- How about?

```
fmap2 :: Functor f => (a->b->c) -> f a -> f b -> f c
  ...
```

- Do not do this!

Partial application and Functors

- Partial application
  - `(++) ::` ?

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" ::` ?
  - `((++) $ "ab") ::` ?

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: ?`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: String`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: String`
- Functors
  - `fmap (++) (Just "ab") ::` ?

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: String`
- Functors
  - `fmap (++) (Just "ab") :: Maybe (String -> String)`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: String`
- Functors
  - `fmap (++) (Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") :: ?`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: String`
- Functors
  - `fmap (++) (Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") :: Maybe (String -> String)`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: String`
- Functors
  - `fmap (++) (Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") ?? Just "cd" :: Maybe String`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: String`
- Functors
  - `fmap (++) (Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") ?? Just "cd" :: Maybe String`
- `?? :: ?`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: String`
- Functors
  - `fmap (++) (Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") ?? Just "cd" :: Maybe String`
- `?? :: Maybe (a -> b) -> Just a -> Just b`

Partial application and Functors

- Partial application
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) "ab" :: String -> String`
  - `((++) $ "ab") :: String -> String`
  - `((++) $ "ab") "cd" :: String`
- Functors
  - `fmap (++) (Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") :: Maybe (String -> String)`
  - `((++) <$> Just "ab") ?? Just "cd" :: Maybe String`
- `?? :: Maybe (a -> b) -> Just a -> Just b`
- Applicative functors: (partial) function application for functors

Applicative Functors (`Control.Applicative`)

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
```

- `Applicative` functors:
  - function application in the context of a functor

Applicative Functors (`Control.Applicative`)

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
```

- `Applicative` functors:
  - function application in the context of a functor
- ```
  h :: a -> b -> c
  h <$> x :: Applicative f => f (b -> c)
  h <$> x <*> y :: Applicative f => f c
  ```
  ```

Applicative style

- `f :: a -> b`
  - `f <$> x`

Applicative style

- `f :: a -> b`
  - `f <$> x`
- `f :: a -> b -> c`
  - `f <$> x <*> y`

Applicative style

- `f :: a -> b`
  - `f <$> x`
- `f :: a -> b -> c`
  - `f <$> x <*> y`
- `f :: a -> b -> c -> d`
  - `f <$> x <*> y <*> z`

Applicative style

- `f :: a -> b`
  - `f <$> x`
- `f :: a -> b -> c`
  - `f <$> x <*> y`
- `f :: a -> b -> c -> d`
  - `f <$> x <*> y <*> z`
- ...

Maybe Applicative

```
instance Applicative Maybe where
  pure = Just
  Nothing  <*> _ = Nothing
  (Just f) <*> x = fmap f x
```

Maybe Applicative

```
instance Applicative Maybe where
  pure = Just
  Nothing  <*> _ = Nothing
  (Just f) <*> x = fmap f x
```

- `(+1) <$> Just 1 == Just 2`

Maybe Applicative

```
instance Applicative Maybe where
  pure = Just
  Nothing  <*> _ = Nothing
  (Just f) <*> x = fmap f x
```

- `(+1) <$> Just 1 == Just 2`
- `(+) <$> Just 1 == Just (1+)`

Maybe Applicative

```
instance Applicative Maybe where
  pure = Just
  Nothing  <*> _ = Nothing
  (Just f) <*> x = fmap f x
```

- `(+1) <$> Just 1 == Just 2`
- `(+) <$> Just 1 == Just (1+)`
- `(+) <$> Just 1 <*> Just 2 == Just 3`

Maybe Applicative

```
instance Applicative Maybe where
  pure = Just
  Nothing  <*> _ = Nothing
  (Just f) <*> x = fmap f x
```

- `(+1) <$> Just 1 == Just 2`
- `(+) <$> Just 1 == Just (1+)`
- `(+) <$> Just 1 <*> Just 2 == Just 3`
- `(+) <$> Nothing <*> Just 2 == Nothing`

Maybe Applicative

```
instance Applicative Maybe where
  pure = Just
  Nothing  <*> _ = Nothing
  (Just f) <*> x = fmap f x
```

- `(+1) <$> Just 1 == Just 2`
- `(+) <$> Just 1 == Just (1+)`
- `(+) <$> Just 1 <*> Just 2 == Just 3`
- `(+) <$> Nothing <*> Just 2 == Nothing`
- `(+) <$> Just 1 <*> Nothing == Nothing`

Maybe Applicative

```
instance Applicative Maybe where
  pure = Just
  Nothing  <*> _ = Nothing
  (Just f) <*> x = fmap f x
```

- `(+1) <$> Just 1 == Just 2`
- `(+) <$> Just 1 == Just (1+)`
- `(+) <$> Just 1 <*> Just 2 == Just 3`
- `(+) <$> Nothing <*> Just 2 == Nothing`
- `(+) <$> Just 1 <*> Nothing == Nothing`
- `(\x y z->x+y+z)<$>Just 1<*>Just 2<*>Just 3=Just 6`

Question 3

What is the result of evaluating the following Haskell expression?

```
(,) <$> (Just 1 <*> Just 2)
```

A `Just 3`

B `(Just 1, Just 2)`

C `Just (1, 2)`

D `This code is incorrect`

Answer 3

- Types:

```
(,) :: a -> b -> (a,b)
fmap (,) :: Functor f => f a -> f (b -> (a,b))
(Just 1 <*> Just 2) :: (Num a,Num (a->b)) => Maybe b
```

Answer 3

- Types:

```
(,) :: a -> b -> (a,b)
fmap (,) :: Functor f => f a -> f (b -> (a,b))
(Just 1 <*> Just 2) :: (Num a,Num (a->b)) => Maybe b
```

- The types do not match

Answer 3

- Types:

```
(,) :: a -> b -> (a,b)
fmap (,) :: Functor f => f a -> f (b -> (a,b))
(Just 1 <*> Just 2) :: (Num a,Num (a->b)) => Maybe b
```

- The types do not match
- Correct code:

```
((,) <$> Just 1) <*> Just 2
```

Answer 3

- Types:

```
(,) :: a -> b -> (a,b)
fmap (,) :: Functor f => f a -> f (b -> (a,b))
(Just 1 <*> Just 2) :: (Num a,Num (a->b)) => Maybe b
```

- The types do not match
- Correct code:

```
((,) <$> Just 1) <*> Just 2
```

- This is the same as:

```
(,) <$> Just 1 <*> Just 2
```

Answer 3

- Types:

```
(,) :: a -> b -> (a,b)
fmap (,) :: Functor f => f a -> f (b -> (a,b))
(Just 1 <*> Just 2) :: (Num a,Num (a->b)) => Maybe b
```

- The types do not match
- Correct code:

```
((,) <$> Just 1) <*> Just 2
```

- This is the same as:

```
(,) <$> Just 1 <*> Just 2
```

- The corrected code results in `Just (1,2)`

List Applicative

```
instance Applicative [] where
  pure x    = [ x ]
  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

List Applicative

```
instance Applicative [] where
  pure x    = [ x ]
  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

- Like a cartesian product

List Applicative

```
instance Applicative [] where
  pure x    = [ x ]
  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

- Like a cartesian product
- Examples

```
Prelude> (++) <$> ["North", "South"] <*> ["east",
"west"]
```

List Applicative

```
instance Applicative [] where
  pure x    = [ x ]
  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

- Like a cartesian product
- Examples

```
Prelude> (++) <$> ["North", "South"] <*> ["east",
"west"]
["Northeast","Northwest","Southeast","Southwest"]
```

List Applicative

```
instance Applicative [] where
  pure x    = [ x ]
  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

- Like a cartesian product
- Examples

```
Prelude> (++) <$> ["North", "South"] <*> ["east",
"west"]
["Northeast","Northwest","Southeast","Southwest"]
Prelude> [(+1), (*4)] <*> [10]
```

List Applicative

```
instance Applicative [] where
  pure x    = [ x ]
  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

- Like a cartesian product
- Examples

```
Prelude> (++) <$> ["North", "South"] <*> ["east",
"west"]
["Northeast","Northwest","Southeast","Southwest"]
Prelude> [(+1), (*4)] <*> [10]
[11,40]
```

IO in Haskell

- `getLine :: IO String`
  - *Operation* inside the `IO` context
  - Reads a line from standard input
  - Results in a `String` in the `IO` context

IO in Haskell

- `getLine :: IO String`
  - *Operation* inside the `IO` context
  - Reads a line from standard input
  - Results in a `String` in the `IO` context
- Referential transparancy:
  - Functions: with the same inputs, always the same outputs
  - In the pure Haskell context, IO is not possible

IO in Haskell

- `getLine :: IO String`
  - *Operation* inside the `IO` context
  - Reads a line from standard input
  - Results in a `String` in the `IO` context
- Referential transparancy:
  - Functions: with the same inputs, always the same outputs
  - In the pure Haskell context, IO is not possible
- Strictly speaking: `getLine` is not a function, but an `IO` action

IO in Haskell

- `getLine :: IO String`
  - *Operation* inside the `IO` context
  - Reads a line from standard input
  - Results in a `String` in the `IO` context
- Referential transparancy:
  - Functions: with the same inputs, always the same outputs
  - In the pure Haskell context, IO is not possible
- Strictly speaking: `getLine` is not a function, but an `IO` action
- IO in Haskell: functions to modify IO actions

IO in Haskell

- `getLine :: IO String`
  - *Operation* inside the `IO` context
  - Reads a line from standard input
  - Results in a `String` in the `IO` context
- Referential transparancy:
  - Functions: with the same inputs, always the same outputs
  - In the pure Haskell context, IO is not possible
- Strictly speaking: `getLine` is not a function, but an `IO` action
- IO in Haskell: functions to modify IO actions
- Haskell evaluates the `IO` action
  - GHCi shows the values wrapped in `IO`

IO in Haskell

- Applicative functors: apply functions in some context

IO in Haskell

- Applicative functors: apply functions in some context
- `IO Applicative`:
  - Allows pure function application in the `IO` context

IO in Haskell

- Applicative functors: apply functions in some context
- `IO Applicative`:
  - Allows pure function application in the `IO` context
- Examples of `IO`:
  - `getChar :: IO Char`
  - `readFile :: FilePath -> IO String`
  - `getLine :: IO String`

IO in Haskell

- Applicative functors: apply functions in some context
- `IO Applicative`:
  - Allows pure function application in the `IO` context
- Examples of `IO`:
  - `getChar :: IO Char`
  - `readFile :: FilePath -> IO String`
  - `getLine :: IO String`
- IO operations beyond “mapping”: `Monad`

IO examples

```
Prelude> (++) <$> getLine <*> getLine
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
cd
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
cd
"cdab"
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
cd
"cdab"
Prelude> (++) <$> (reverse <$> getLine) <*>
getLine
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
cd
"cdab"
Prelude> (++) <$> (reverse <$> getLine) <*>
getLine
ab
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
cd
"cdab"
Prelude> (++) <$> (reverse <$> getLine) <*>
getLine
ab
cd
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
cd
"cdab"
Prelude> (++) <$> (reverse <$> getLine) <*>
getLine
ab
cd
"bacd"
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
cd
"cdab"
Prelude> (++) <$> (reverse <$> getLine) <*>
getLine
ab
cd
"bacd"
Prelude> (++) <$> getLine <*> (pure "!")
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
cd
"cdab"
Prelude> (++) <$> (reverse <$> getLine) <*>
getLine
ab
cd
"bacd"
Prelude> (++) <$> getLine <*> (pure "!")
ab
```

IO examples

```
Prelude> (++) <$> getLine <*> getLine
ab
cd
"abcd"
Prelude> (flip (++)) <$> getLine <*> getLine
ab
cd
"cdab"
Prelude> (++) <$> (reverse <$> getLine) <*>
getLine
ab
cd
"bacd"
Prelude> (++) <$> getLine <*> (pure "!")
ab
"ab!"
```

Alternative

Alternative typeclass

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  some :: f a -> f [a]
  many :: f a -> f [a]
```

- Required definitions: `empty` and `<|>`

Alternative typeclass

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  some :: f a -> f [a]
  many :: f a -> f [a]
```

- Required definitions: `empty` and `<|>`
- Monoid on Applicative functors
  - `empty` defines the identity
  - `<|>` defines the binary operation

Alternative typeclass

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  some :: f a -> f [a]
  many :: f a -> f [a]
```

- Required definitions: `empty` and `<|>`
- Monoid on Applicative functors
  - `empty` defines the identity
  - `<|>` defines the binary operation
- Of kind `* -> *` (Monoid: *)

Alternative typeclass

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  some :: f a -> f [a]
  many :: f a -> f [a]
```

- Required definitions: `empty` and `<|>`
- Monoid on Applicative functors
  - `empty` defines the identity
  - `<|>` defines the binary operation
- Of kind `* -> *` (Monoid: *)
- Used in ParSec:
  - `<|>`: parser combinator shown in last lecture
  - `many`:
    - Grammar: `('a')*`
    - Haskell: `many (char 'a')`
  - `some`:
    - Grammar: `('a')+`
    - Haskell: `some (char 'a')`

List Alternative

```
instance Alternative [] where
  empty = []
  (<|>) = (++)
```

```
*Lec6> (+) <$> ([1,2,3] <|> [4]) <*> [10,100]
```

List Alternative

```
instance Alternative [] where
  empty = []
  (<|>) = (++)
```

```
*Lec6> (+) <$> ([1,2,3] <|> [4]) <*> [10,100]
[11,101,12,102,13,103,14,104]
```

List Alternative

```
instance Alternative [] where
  empty = []
  (<|>) = (++)
```

```
*Lec6> (+) <$> ([1,2,3] <|> [4]) <*> [10,100]
[11,101,12,102,13,103,14,104]
*Lec6> (pure (+) <|> pure (*)) <*> [1,2,3] <*>
[10,100]
```

List Alternative

```
instance Alternative [] where
  empty = []
  (<|>) = (++)
```

```
*Lec6> (+) <$> ([1,2,3] <|> [4]) <*> [10,100]
[11,101,12,102,13,103,14,104]
*Lec6> (pure (+) <|> pure (*)) <*> [1,2,3] <*>
[10,100]
[11,101,12,102,13,103,10,100,20,200,30,300]
```

IO Alterative

- `a <|> b`
  - Execute action `a`
  - When this results in an IO exception: execute `b`

IO Alterative

- `a <|> b`
  - Execute action `a`
  - When this results in an IO exception: execute `b`
- Examples (file `a` exists, file `b` does not)

IO Alterative

- `a <|> b`
  - Execute action `a`
  - When this results in an IO exception: execute `b`
- Examples (file `a` exists, file `b` does not)

```
Lec6*> length <$> (readFile "a" <|> (pure ""))
```

IO Alterative

- `a <|> b`
  - Execute action `a`
  - When this results in an IO exception: execute `b`
- Examples (file `a` exists, file `b` does not)

```
Lec6*> length <$> (readFile "a" <|> (pure ""))
20
```

IO Alterative

- `a <|> b`
  - Execute action `a`
  - When this results in an IO exception: execute `b`
- Examples (file `a` exists, file `b` does not)

```
Lec6*> length <$> (readFile "a" <|> (pure ""))
20
Lec6*> length <$> (readFile "b" <|> (pure ""))
```

IO Alterative

- `a <|> b`
  - Execute action `a`
  - When this results in an IO exception: execute `b`
- Examples (file `a` exists, file `b` does not)

```
Lec6*> length <$> (readFile "a" <|> (pure ""))
20
Lec6*> length <$> (readFile "b" <|> (pure ""))
0
```

IO Alterative

- `a <|> b`
  - Execute action `a`
  - When this results in an IO exception: execute `b`
- Examples (file `a` exists, file `b` does not)

```
Lec6*> length <$> (readFile "a" <|> (pure ""))
20
Lec6*> length <$> (readFile "b" <|> (pure ""))
0
Lec6*> length <$> (readFile "a" <|> readFile "b")
```

IO Alterative

- `a <|> b`
  - Execute action `a`
  - When this results in an IO exception: execute `b`
- Examples (file `a` exists, file `b` does not)

```
Lec6*> length <$> (readFile "a" <|> (pure ""))
20
Lec6*> length <$> (readFile "b" <|> (pure ""))
0
Lec6*> length <$> (readFile "a" <|> readFile "b")
20
```

Monoids

Monoids in mathematics (abstract algebra)

Definition (Monoid)

A set S with an associative operation $\circ : S \times S \to S$ is a *Monoid* when it satisfies the axioms:

- Associativity: For all $a, b, c \in S$ it holds $(a \circ b) \circ c = a \circ (b \circ c)$
- Identity element: There exists an $e \in S$ such that for all $a \in S$ it holds $a \circ e = e \circ a = a$

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                -- identity element e
  mappend :: a -> a -> a -- associative operation
  mconcat :: [a] -> a
```

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                -- identity element e
  mappend :: a -> a -> a -- associative operation
  mconcat :: [a] -> a
```

- Instances require `mempty` and `mappend`

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                 -- identity element e
  mappend :: a -> a -> a -- associative operation
  mconcat :: [a] -> a
```

- Instances require `mempty` and `mappend`
- Expresses combining two values(/objects) to a single

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                 -- identity element e
  mappend :: a -> a -> a -- associative operation
  mconcat :: [a] -> a
```

- Instances require `mempty` and `mappend`
- Expresses combining two values(/objects) to a single
- Infix operator <>
  - `mappend x y == x <> y`

Monoids in Haskell (`Data.Monoid`)

```haskell
class Monoid m where
  mempty :: a                -- identity element e
  mappend :: a -> a -> a -- associative operation
  mconcat :: [a] -> a
```

- Instances require `mempty` and `mappend`
- Expresses combining two values(/objects) to a single
- Infix operator <>
  - `mappend x y == x <> y`
- `mconcat [x1, x2, ...] == x1 <> x2 <> ...`

Monoid example: []

```
instance Monoid [a] where
  mempty = []
  mappend xs ys = xs ++ ys
```

Monoid example: []

```
instance Monoid [a] where
  mempty = []
  mappend xs ys = xs ++ ys
```

```
*Lec6> "abc" <> "def"
```

Monoid example: []

```
instance Monoid [a] where
  mempty = []
  mappend xs ys = xs ++ ys
```

```
*Lec6> "abc" <> "def"
"abcdef"
```

Monoid example: []

```
instance Monoid [a] where
  mempty = []
  mappend xs ys = xs ++ ys
```

```
*Lec6> "abc" <> "def"
"abcdef"
*Lec6> mconcat [[1,2,3] , [], [7..10]]
```

Monoid example: []

```
instance Monoid [a] where
  mempty = []
  mappend xs ys = xs ++ ys
```

```
*Lec6> "abc" <> "def"
"abcdef"
*Lec6> mconcat [[1,2,3] , [], [7..10]]
[1,2,3,7,8,9,10]
```

Monoid example: Sum

```haskell
-- The definition in GHC is slightly different!
data Sum a = Sum { getSum :: a }
           deriving Show
```

Monoid example: Sum

```haskell
-- The definition in GHC is slightly different!
data Sum a = Sum { getSum :: a }
           deriving Show

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

Monoid example: Sum

```haskell
 -- The definition in GHC is slightly different!
data Sum a = Sum { getSum :: a }
           deriving Show

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

```
*Lec6> Sum 10 <> Sum 20
```

Monoid example: Sum

```haskell
-- The definition in GHC is slightly different!
data Sum a = Sum { getSum :: a }
           deriving Show

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

```
*Lec6> Sum 10 <> Sum 20
Sum {getSum = 30}
```

Monoid example: Sum

```haskell
-- The definition in GHC is slightly different!
data Sum a = Sum { getSum :: a }
           deriving Show

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

```
*Lec6> Sum 10 <> Sum 20
Sum {getSum = 30}
*Lec6> vs = [Sum 4, Sum 10, Sum 5, Sum 1]
```

Monoid example: Sum

```haskell
-- The definition in GHC is slightly different!
data Sum a = Sum { getSum :: a }
           deriving Show

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

```
*Lec6> Sum 10 <> Sum 20
Sum {getSum = 30}
*Lec6> vs = [Sum 4, Sum 10, Sum 5, Sum 1]
*Lec6> getSum (mconcat vs)
```

Monoid example: Sum

```haskell
-- The definition in GHC is slightly different!
data Sum a = Sum { getSum :: a }
           deriving Show

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

```
*Lec6> Sum 10 <> Sum 20
Sum {getSum = 30}
*Lec6> vs = [Sum 4, Sum 10, Sum 5, Sum 1]
*Lec6> getSum (mconcat vs)
20
```

Monoid example: Sum

```haskell
 -- The definition in GHC is slightly different!
data Sum a = Sum { getSum :: a }
           deriving Show

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

```
*Lec6> Sum 10 <> Sum 20
Sum {getSum = 30}
*Lec6> vs = [Sum 4, Sum 10, Sum 5, Sum 1]
*Lec6> getSum (mconcat vs)
20
*Lec6> (getSum .  mconcat .  map Sum) [1..10]
```

Monoid example: Sum

```haskell
 -- The definition in GHC is slightly different!
data Sum a = Sum { getSum :: a }
           deriving Show

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

```
*Lec6> Sum 10 <> Sum 20
Sum {getSum = 30}
*Lec6> vs = [Sum 4, Sum 10, Sum 5, Sum 1]
*Lec6> getSum (mconcat vs)
20
*Lec6> (getSum .  mconcat .  map Sum) [1..10]
55
```

Maybe Monoid

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing  `mappend` x        =  x
  x        `mappend` Nothing  =  x
  (Just x) `mappend` (Just y) = Just (x `mappend` y)
```

Monoid wrappers

Wrapper	wrapped type	identity	binary operator
`All`	`Bool`	`True`	`&&`
`Any`	`Bool`	`False`	`||`
`Dual`	`[]`	`[]`	`flip (++)`
`First`	`Maybe a`	`Nothing`	(non elementary)
`Last`	`Maybe a`	`Nothing`	(non elementary)
`Product`	`Num a => a`	`1`	`*`
`Sum`	`Num a => a`	`0`	`+`

Monoid wrappers

Wrapper	wrapped type	identity	binary operator
`All`	`Bool`	`True`	`&&`
`Any`	`Bool`	`False`	`||`
`Dual`	`[]`	`[]`	`flip (++)`
`First`	`Maybe a`	`Nothing`	(non elementary)
`Last`	`Maybe a`	`Nothing`	(non elementary)
`Product`	`Num a => a`	`1`	`*`
`Sum`	`Num a => a`	`0`	`+`

Use `map` to wrap the values

Foldable

Foldable

- Foldable type class: things you can fold over

Foldable

- Foldable type class: things you can fold over
- Required: either `foldMap` *or* `foldr` (easiest)

Foldable

- Foldable type class: things you can fold over
- Required: either `foldMap` *or* `foldr` (easiest)
- Provides: `foldr`, `foldl`, `foldMap`, `length`, `sum` ...

Foldable type class

```
class Foldable t where
  fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldl :: (b -> a -> b) -> b -> t a -> b
  foldr1 :: (a -> a -> a) -> t a -> a
  foldl1 :: (a -> a -> a) -> t a -> a
  toList :: t a -> [a]
  null :: t a -> Bool
  length :: t a -> Int
  elem :: Eq a => a -> t a -> Bool
  maximum :: Ord a => t a -> a
  minimum :: Ord a => t a -> a
  sum :: Num a => t a -> a
  product :: Num a => t a -> a
  ...
```

Self study

- This lecture: “Advanced type classes”
  - Topics: `Monoid`, `Foldable` and `Applicative`
  - Books:
    - Learn you a Haskell (Lipovaca): Chapter 8 (Functor), Chapter 11 (Applicative, Monoid, ,Foldable)
    - Programming in Haskell (Hutton): Chapter 12.1 (Functor), Chapter 12.2 (Applicative), Chapter 14.1 (Monoid), Chapter 14.2 (Foldable)

Remaining lectures

- 14-05: Next lecture: Review material block 2 and block 3
  - More `Applicative` instances
  - Some background on QuickCheck
  - VoxVote

Remaining lectures

- 14-05: Next lecture: Review material block 2 and block 3
  - More `Applicative` instances
  - Some background on QuickCheck
  - VoxVote

- 23-05: Introduction “Project Functional Programming”
  - Project: covers all material
  - Lecture to introduce the assignment

Remaining lectures

- 14-05: Next lecture: Review material block 2 and block 3
  - More `Applicative` instances
  - Some background on QuickCheck
  - VoxVote
- 23-05: Introduction “Project Functional Programming”
  - Project: covers all material
  - Lecture to introduce the assignment
- 13-06: Q&A Functional Programming
  - Last opportunity to ask questions
  - Send me an e-mail before 12-06 with requests for additional explanations of the material

Remaining lectures

- 14-05: Next lecture: Review material block 2 and block 3
  - More `Applicative` instances
  - Some background on QuickCheck
  - VoxVote
- 23-05: Introduction “Project Functional Programming”
  - Project: covers all material
  - Lecture to introduce the assignment
- 13-06: Q&A Functional Programming
  - Last opportunity to ask questions
  - Send me an e-mail before 12-06 with requests for additional explanations of the material
- 14-06: Exam Functional Programming