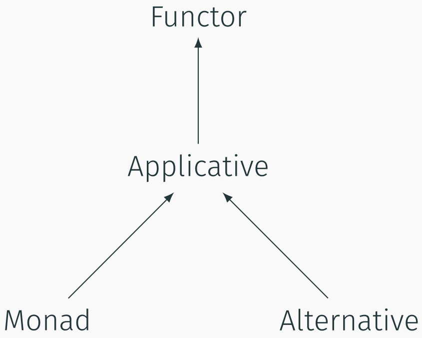
Functional Programming
Lecture 12

Rostislav Horčík

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

Applicative functors

Functor subclasses



Functors

Functor instances allow to lift a unary map to the functorial context.

```haskell
fmap :: (a -> b) -> f a -> f b
```

`(+2) :: Num a => a -> a` lifts to
`fmap (+2) :: (Num b, Functor f) => f b -> f b`

But we cannot lift binary `(+) :: Num a => a -> a -> a`

```haskell
Just 3 <+> Just 5
```

If we lift `(+)` by `fmap`,

```haskell
fmap (+) :: (Num a, Functor f) => f a -> f (a -> a)
```

```haskell
fmap (+) (Just 2) :: Num a => Maybe (a -> a)
```

Instances

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

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

pure (,) <*> [1,2,3] <*> ['a','b','c']
```

Monadic parsing

Parser

Parser is a program taking an input string and converting it into a data structure containing all the information encoded in the input string. E.g. a source file is coverted into AST.

```
data Expr a = Val a
    | Var String
    | Add [Expr a]
    | Mul [Expr a] deriving Eq
```

```
"(4 * (5 + 7 + x))"
```

is converted into

```
Mul [Val 4, Add [Val 5,Val 7,Var "x"]]
```

Grammar

```
<expr> -> <space>* <expr'> <space>*
<expr'> -> <var>
         | <val>
         | <add>
         | <mul>

<var> -> <lower> <alphanum>*
<val> -> <int> "." <digit>+ | <int>
<int> -> "-" <digit>+ | <digit>+

<add> -> "(" <expr> ("+" <expr>)+ ")"
<mul> -> "(" <expr> ("*" <expr>)+ ")"
```

Type constructor parser

`Parser a` is a function taking a string and returning a parsed value of type `a` together with the remaining unused string. The parsing may fail.

```
newtype Parser a =
P { parse :: String -> Maybe (a, String) }
```

```
item :: Parser Char
item = P (\inp -> case inp of
                    "" -> Nothing
                    (x:xs) -> Just (x,xs))
```

Functor instance

```
instance Functor Parser where
  -- fmap :: (a -> b) -> Parser a -> Parser b
  fmap f p = P (\inp ->
    case parse p inp of
      Nothing -> Nothing
      Just (v,out) -> Just (f v, out))
```

```
> parse (fmap (=='c') item) "cde"
Just (True,"de")
```

```
> parse (fmap (=='c') item) "ade"
Just (False,"de")
```

Applicative instance

```
instance Applicative Parser where
--(<*>) :: Parser (a -> b) -> Parser a -> Parser b
  pg <*> px = P (\inp ->
    case parse pg inp of
      Nothing -> Nothing
      Just (g,out) -> parse (fmap g px) out)

  pure v = P (\inp -> Just (v,inp))
```

```
> parse (pure (/=) <*> item <*> item) "abc"
Just (True,"c")
```

```
> parse (pure (/=) <*> item <*> item) "aac"
Just (False,"c")
```

Monad instance

```
instance Monad Parser where
--(>>=) :: Parser a -> (a -> Parser b) -> Parser b
  p >>= f = P (\inp ->
    case parse p inp of
      Nothing -> Nothing
      Just (v,out) -> parse (f v) out)
```

```
> parse (item >>= \c ->
    if c == 'a' then item else return ' ') "abc"
Just ('b',"c")
```

```
> parse (item >>= \c ->
    if c == 'a' then item else return ' ') "xbc"
Just (' ',"bc")
```

Alternative instance

```
instance Alternative Parser where
-- empty :: Parser a
empty = P (\_ -> Nothing)

-- (<|>) :: Parser a -> Parser a -> Parser a
p <|> q = P (\inp ->
    case parse p inp of
      Nothing -> parse q inp
      Just (v,out) -> Just (v,out))
```

```
> parse empty "abc"
Nothing
```

```
> parse (item <|> return 'x') ""
Just ('x',"")
```

Building parsers

```haskell
sat :: (Char -> Bool) -> Parser Char
sat pr = item >>= \x -> if pr x then return x
                         else empty

alphaNum :: Parser Char
alphaNum = sat isAlphaNum


char :: Char -> Parser Char
char c = sat (== c)

string :: String -> Parser String
string [] = return []
string (x:xs) = char x
                >> string xs
                >> return (x:xs)
```

many and some

Automatically defined for instances of `Alternative`

```
many :: f a -> f [a]
some :: f a -> f [a]

many p = some p <|> pure []
some p = pure (:) <*> p <*> many p
```

`many p`, `some p` — both perform repeatedly parser `p` until it fails and returns a list of its results.

`many p` — always succeeds, might return the empty list

`some p` — succeeds if `p` succeeds at least once

```
parse (some (char 'a')) "aaabc"
```

Homework assignment 4 — Parser of λ-programs

Aim: To practice monadic parsing in Haskell, together with HW3 build a complete λ-calculus interpreter

```
0 := (\s.(\z.z))
S := (\w.(\y.(\x.(y ((w y) x)))))
1 := (S 0)
2 := (S 1)
((2 S) 1)
```

Points: 13

Deadline: in 3 weeks (May 26)

Penalty: after deadline -1 points every day (at most -12)

Description: all details can be found in CW

Summary

- `Applicative` is a type subclass of `Functor` allowing to lift *n*-ary maps to the functorial context.
- `Parser` is a type constructor returning a function of type `String -> Maybe (a, String)`.
- We defined its `Monad` instance.
- It allows to build more complex parsers out of the simple ones.
- `Alternative` is a type subclass of `Applicative`.
- It allows to choice between several parsers.
- It implements `many p` and `some p` behaving like `p*` and `p+` respectively.