LECTURE 09

Theory and Design of PL (CS 538)

February 19, 2020

GOING BEYOND FUNCTOR

MAPPING OVER ONE THING

- A function for shouting things:

```
shout :: String -> String
shout = toUpper

-- shout "hello" === "HELLO"
```

- Mapping this function is easy:

```
shoutMaybe :: Maybe String -> Maybe String
shoutMaybe = fmap shout

shoutList :: [String] -> [String]
shoutList = fmap shout
```

MAPPING OVER MORE THINGS

- A function for shouting two things?!

```
shout2 :: String -> String -> String
shout2 x y = (toUpper x) ++ " " ++ (toUpper y)

-- shout2 "hello" "world" === "HELLO WORLD"
```

- This is OK, but how do we map this thing?

```
shout2Maybe :: Maybe String -> Maybe String -> Maybe String
shout2Maybe = ???
```

THE UGLY WAY

```
shout2Maybe :: Maybe String -> Maybe String -> Maybe String
shout2Maybe Nothing _ = Nothing
shout2Maybe _ Nothing = Nothing
shout2Maybe (Just x) (Just y) = Just (shout2 x y)
```

- Seems like a lot of trouble just to use `shout2`
  - `shout3Maybe`, `shout100Maybe`, ...?

AN INITIAL TRY

- We know Maybe is a Functor, so let's try `fmap`
- We can map over the first argument, but then stuck:

```
shout2Maybe :: Maybe String -> Maybe String -> Maybe String
shout2Maybe mx my = let shoutFirst = fmap shout2 mx in
                      -- shoutFirst :: Maybe (String -> String)
                      -- ... now what?
```

- Apply "Maybe (String -> String)" to "Maybe String"?

SOLUTION: APPLICATIVE

- We can solve this problem by extending Functor

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

LET'S DEFINE FOR MAYBE

- As always: follow the types...

```
instance Applicative Maybe where
    -- pure :: a -> Maybe a
    pure x = Just x

    -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
    Nothing <*> _ = Nothing
    _ <*> Nothing = Nothing
    (Just f) <*> (Just x) = Just (f x)
```

REVISITING SHOUT...

- Now: we can define `shout2Maybe`

```
shout2Maybe :: Maybe String -> Maybe String -> Maybe String
shout2Maybe mx my = let shoutFirst = fmap shout2 mx in
                    shoutFirst <*> my
```

- Cleaning things up a bit more...

```
shout2Maybe mx my = shout2 <$> mx <*> my
-- associates left: (shout2 <$> mx) <*> my
```

APPLICATIVE LAWS

- Laws are more complicated (don't memorize)

```
-- 1. identity
pure id <*> v === v

-- 2. homomorphism
pure f <*> pure x === pure (f x)

-- 3. interchange
u <*> pure y === pure ($ y) <*> u

-- 4. composition
pure (.) <*> u <*> v <*> w === u <*> (v <*> w)
```

EXAMPLE: LISTS

- Let's write an applicative instance for list
- Follow the types…

```
instance Applicative ([]) where
    -- pure :: a -> [a]
    pure x = [x]

    -- (<*>) :: [a -> b] -> [a] -> [b]
    [] <*> _ = []
    (f:fs) <*> xs = fmap f xs ++ fs <*> xs

    -- associates: (fmap f xs) ++ (fs <*> xs)
```

- Apply each function to every element, then collect

ANOTHER WAY: LISTS

- There's another, less obvious instance...

```
instance Applicative ([]) where
    -- pure :: a -> [a]
    pure x = x : pure x  -- infinite list of x

    -- (<*>) :: [a -> b] -> [a] -> [b]
    fs <*> xs = zipWith ($) fs xs
```

- Apply each function to *one* element
  - Relies on Haskell's lazy evaluation...

WHAT IS PARSING?

TURN UNSTRUCTURED DATA INTO STRUCTURED DATA

- Data is stored and transmitted as plain text
- Structure indicated by special characters
  - Line breaks and whitespace
  - Commas and other punctuation
  - Parentheses, matching open/close tags
- For programs to use this data, need to convert from "list of characters" to something more structured

EXAMPLES EVERYWHERE

- Compilation
  - Source code transformed to AST and compiled
- Compression
  - Files converted to and from compressed form
- Networking
  - HTTP headers, data feeds, API requests, ...
- Logging
  - System monitoring, error logs, ...

PARSING IS ANNOYING

- Theoretically well-studied, many algorithms
  - LL, LR, Earley, CYK, shift-reduce, Packrat, …
- Writing parsers is tedious, often use *parser generators*
  - Write grammar in a special language, get a parser
  - ANTLR, Bison, Yacc, …
- Parser language drawbacks
  - Complex and hard to read: error prone!
  - Not a full-featured language

BUILDING A PARSER IN HASKELL

PLAN FOR NEXT FEW DAYS

- Build a small library for parsers in Haskell
- Good example of a *domain-specific language* (DSL)
  - Small, special-purpose language
  - Strength of Haskell and FP

> *HW3: Extend parser with more features*

MAIN PARSER TYPE

- Goal: parse a string into a type `a`
- We call `(a, String)` a *parse* (result)
  - First component: output of parser
  - Second component: rest of string ("" is done)
- Parser: function from string to Maybe parse

```
data Parser a = MkParser (String -> Maybe (a, String))
```

RUNNING THE PARSER

1. Plug in the input string and run function

```
runParser :: Parser a -> String -> Maybe (a, String)
runParser (MkParser parseFn) input = parseFn input
```

RUNNING THE PARSER

2. Filter out parses that don’t consume whole string

```
getParse :: Parser a -> String -> Maybe a
getParse parser input = case runParser parser input of
  Nothing          -> Nothing   -- Parser couldn't parse anything
  Just (val, "") -> Just val   -- Got result, finished string
  Just _           -> Nothing   -- Got result, but leftover string
```

DESIGN PHILOSOPHY

1. First: tiny, building-block parsers
    - Will seem really limited, almost boring
2. Next: basic ways to combine parsers
    - Choice, sequencing, …
3. Then: complex ways to combine parsers
    - Repetition, separation, …

> *Build big parsers out of simpler parsers!*

SOME SIMPLE PARSERS

- Empty string: don't consume any input

```
emptyP :: Parser String
emptyP = MkParser $ \str -> Just ("", str)
```

- Parse one of any character

```
itemP :: Parser Char
itemP = MkParser $ \str ->
  case str of
    []     -> Nothing
    (c:cs) -> Just (c, cs)
```

MORE SIMPLE PARSERS

- Parse one of some kind of character

```haskell
charSatP :: (Char -> Bool) -> Parser Char
charSatP predicate = MkParser $ \str ->
  case str of
    []      -> Nothing
    (c:cs) -> if predicate c then Just (c, cs) else Nothing

spaceP :: Parser Char
spaceP = charSatP isSpace

digitP :: Parser Char
digitP = charSatP isDigit

charP  :: Char -> Parser Char
charP c = charSatP (== c)
```

APPLICATIVE PARSING

THE STORY SO FAR

- Parser: input String to parsed value, rest of String
- We have: basic parsers (one char, digit, space, ...)
- Needed: parser transformers
  - Take parser, change/process "parsed value"
- Needed: parser combinators
  - Combine parsers into larger parsers

PARSER TRANSFORMERS

- Wanted: function with the following type

```
trans :: (a -> b) -> Parser a -> Parser b
```

- Looks familiar? Let's define a Functor instance:

```
instance Functor Parser where
  -- fmap :: (a -> b) -> Parser a -> Parser b
  fmap f par = MkParser $ \str ->
    case runParser par str of
      Nothing -> Nothing
      Just (val, str') -> Just (f val, str')
```

PARSER COMBINATORS: APPLICATIVE

- Applicative will let us combine multiple parsers:

```
instance Applicative Parser where
  -- pure :: a -> Parser a
  pure x = MkParser $ \str -> Just (x, str)

  -- (<*>) :: Parser (a -> b) -> Parser a -> Parser b
  parF <*> parA = MkParser $ \str ->
    case runParser parF str of                      -- run first
      Nothing -> Nothing                            -- first failed
      Just (f, str') ->                             -- first OK
        case runParser parA str' of                 -- run second
          Nothing -> Nothing                        -- second failed
          Just (v, str'') -> Just (f v, str'')      -- second OK
```

- Kind of sequencing: feed `str'` to second parser