



# Advanced Functional Programming

## Week 8 – Backtracking during Parsing, Applicative Functors, Monad Transformers

René Thiemann

Department of Computer Science

### Details on Backtracking during Parsing

### Last Week

- context free grammars
- parser combinators, example: Parsec library
  - several primitives to read a single char, e.g., `char`, `anyOf`, `noneOf`, `space`, `satisfy`, `eof`
  - combinators to combine parsers, e.g., `many`, `many1`, `sepBy`, `endBy`
  - `p1 <|> p2` and `try p1` are used for non-determinism and back-tracking
    - if `p1` succeeds (`Ok`), then `p1 <|> p2` and `try p1` succeed
    - if `p1` fails after consuming some input (`F+`), then `p1 <|> p2` fails (`F+`)
    - if `p1` fails without consuming input (`F0`), then `p1 <|> p2` switches to `p2`
    - if `p1` fails (`F+` or `F0`) then `try p1` fails with `F0`  
(backtrack to position in input stream, when `p1` was started)
    - `p1 <|> p2` tries `p2` only if `p1` fails with `F0`
    - `try p1 <|> p2` tries `p2` only if `p1` fails

RT (DCS @ UIBK)

Week 8

2/30

### Simple ARI Parser (Demo08\_Parser\_ARI\_Do\_Blocks)

```
lexeme p = do
  a <- p
  spaces
  return a

identChar = noneOf " \t\n();;"
identifier = lexeme $ many1 identChar

term = variable <|> funapp

variable = do
  i <- identifier
  return $ Var i

charS c = do
  _ <- lexeme (char c)
  return ()
```

```
funapp = do
  charS '('
  f <- identifier
  ts <- many term
  charS ')'
  return $ Fun f ts

rule = do
  try $ do
    charS '('
    exactlyS "rule"
    l <- term
    r <- term
    charS ')'
    return (l,r)

exactlyS s = lexeme $ try $ do
  _ <- string s
  notFollowedBy identChar <?> "..." ++ s
```

## Explanations

- `lexeme`
  - `lexeme p` has the same behavior as `p`, except that trailing white space is removed
  - invariant: all parsers remove trailing white space
  - advantage: later parsers can always assume that there is no leading white space
  - only exception: the main parser has to once remove leading white space
- `charS` is just a version of `char` that strips trailing white space and does not return the resulting character
- in the `rule` parser, `try` is used to backtrack to the beginning, if the initial part is not of shape  $(\_)*rule$  where `rule` cannot be extended into a longer identifier
- to ensure the latter we use `exactlyS`, which basically is using `string "rule"` followed by the combinator `notFollowedBy`; this combinator usage enforces that no identifier character is present after `"rule"`
  - `"rule a"` is accepted by `exactlyS "rule"`, and one jumps to the beginning of `"a"`
  - `"rules a"` is not accepted by `exactlyS "rule"`, and one jumps back to the beginning of the text, complaining about `"...rule"`

## When to Use Try

- with `try` and `<|>` one can easily write inefficient parsers
- `try` gives rise to **backtracking**, and this can become expensive
- example: detect cases of  $(ab)^* \cup \{a, b\}^*c$   
`pQuadratic = try (string "ab" >> pQuadratic)`  
`<|> (eof >> return "(ab)^*")`  
`<|> (many (oneOf "ab") >> string "c" >> eof >> return "end in c")`
- solution: close try-blocks, as soon as the applicable rule has been determined
- equivalent parser with linear runtime where only the first `try` has been changed  
`pLinear = (try (string "ab") >> pLinear)`  
`<|> (eof >> return "(ab)^*")`  
`<|> (many (oneOf "ab") >> string "c" >> eof >> return "end in c")`
- reason for linear time is the behavior of `<|>`
  - whenever `p1` in `p1 <|> p2` consumes at least one character, then `p2` is not tried
  - consequently, if input starts with `"ab"`, then other alternatives are not tried in `pLinear`

## Example: Small Try-Blocks in ARI Parser

- have a look at the rule parser again  

```
rule = do
  try $ do
    charS '('
    exactlyS "rule"
  l <- term
  r <- term
  charS ')'
  return (l,r)
```
- `try` is closed after keyword `"rule"` has been detected
  - hence, after reading `(rule_` the applied parser is fixed
- one can define similar parsers `p1`, ..., `pn`, for each function symbol in a TRS
  - hence, `choice [p1, ..., pn]` will quickly select the correct parser for `(fNameI t1 ... tk)`, namely after reading `(fNameI_`, without major backtracking

## Applicative Functors

## Applicative Functors, Applicative Style

- have a look at an excerpt of a previous parser

```
parse1 = many (oneOf "ab") >> string "c" >> return "end in c"
```
- here, >> is used to write the parser more succinctly
- alternative without >>

```
parse2 = do
  _ <- many (oneOf "ab")
  _ <- string "c"
  _ <- eof
  return "end in c"
```
- observation: we often invoke several parsers, but only some of them contribute to the parsed result
- >> is only one possible way to combine results: throw away result of left parser
- aim: more flexible combinations
- solution: use **applicative functors** and **applicative style**

## Applicative Functors, Difference to Functors

- known: monads have more structure than functors
- **applicative functors** are between monads and functors

```
class Functor f => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
  pure  :: a -> f a
```
- applicative functors are stronger than **Functors**: it is possible to **lift  $n$ -ary functions to a sequence of  $n$  elements** of an applicative functor, which is not possible with ordinary functors
  - $n = 2$ 

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 g x y = (pure g <*> x) <*> y
```

    - note the partial application: `pure g <*> x :: f (b -> c)`
    - since <\*> associates to the left, one just writes `pure g <*> x <*> y`
  - arbitrary  $n$ : `pure g <*> x1 <*> x2 <*> ... <*> xn`

## Applicative Functors: Laws

- laws
  - `pure id <*> v = v` (identity)
  - `pure g <*> pure x = pure (g x)` (homomorphism)
  - `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)` (composition)
  - `u <*> pure y = pure ($ y) <*> u` (interchange)
- consequence: `fmap g x = pure g <*> x`  
so `fmap` can be implemented via `pure` and `<*>`
- note the similarity and difference of type of `fmap`, `<$>` and `<*>`

```
(<$>) :: f (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
```
- as we have seen, this small change is sufficient to allow arbitrary liftings of  $n$ -ary functions into the applicative functor

## Towards Programming in Applicative Style

- we have already seen that sequences of <\*> can combine results
- sometimes it is helpful to disregard some of the results, while still having the effect of the functor
- therefore, there are several combinators, all with fixity declaration `infixl 4`
- in general, operators with a one sided arrow symbol > use only the result from that side
- all types with \* assume **Applicative f**, all with \$ assume **Functor f**

```
(<*>) :: f (a -> b) -> f a -> f b
(<*)  :: f a -> f b -> f a
(*>) :: f a -> f b -> f b
(<$>) :: (a -> b) -> f a -> f b
(<$)  :: a -> f b -> f a
```
- example implementations

```
(<$) = fmap . const
u (*>) v = (id <$ u) <*> v
```

## Programming in Applicative Style

- combine the combinators of previous slide for more succinct code
- once one gets familiar with these, this does not hinder readability
- example: live demo to switch from Demo08\_Parser\_ARI\_Do\_Blocks to Demo08\_Parser\_ARI\_Applicative
- example explanation of function application parser:  

```
funapp = Fun <$> (charS '(' *> identifier) <*> many term <*> charS ')'
```

  - `charS '(' *> identifier` consumes (`fName`
    - since we are not interested in the open parenthesis, the result of the left parser is ignored by `*>`
    - result of parser will be just `fName`
  - `Fun <$> (charS '(' *> identifier)`
    - parsing is identical, but result will now be `Fun fName`, a partially applied constructor
  - `Fun <$> (charS '(' *> identifier) <*> many term`
    - additionally, many terms `ts` are parsed and result will be the term `Fun fName ts`
  - `Fun <$> (charS '(' *> identifier) <*> many term <*> charS ')'`
    - a closing parenthesis is parsed, but this has no impact on the result, since `<*>` looks to the left

## Applicative Functors and Monads

- every `Monad` is an applicative functor  

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

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure
```
- monads are stronger than applicative functors
  - `(*>) = (>>)`, `a1 <*> a2 = a1 >>= (\ f -> a2 >>= (\ x -> return (f x)))`
  - consider a computation involving  $n$  (monadic or functor) values
    - for applicative functors, the computation of `f <$> v1 <*> v2 <*> ... <*> vn` is possible, but each `vi` is computed **independently**, i.e., `vi` may not look into the results of `v1, ..., vi-1`
    - this is in contrast to monads, where this is possible:  

```
do { x1 <- v1; x2 <- v2 x1; ... xn <- vn x1 ... (xn - 1);
    return $ f x1 ... xn }
```
- example where monads are required: parser for terms can depend on parsed signature

## Applicative Functors and Monads (Continued)

- sometimes monad laws are too restrictive, if one just wants to have an applicative functor
- example: collect errors during computations
- monad laws enforce the following implementation of `>>=`, so that an error in the second argument of `>>` is ignored, if first argument results in error

```
instance Monad (Either e) where
  return = Right
  Left e  >>= _ = Left e           -- Left e1 >> _ = Left e1
  Right x >>= f = f x
```

- just requiring an applicative functor permits an implementation that collects errors

```
instance Monoid e => Applicative (Either e) where
  pure = Right
  Left e1 <*> Left e2 = Left (e1 <> e2) -- Left e1 *> Left e2
  Right f <*> Right x = Right $ f x    -- = Left $ e1 <> e2
  Left e1 <*> _       = Left e1
  _ <*> Left e2       = Left e2
```

## Monad Transformer

## Using Several Monads at Once: Monad Transformer

- sometimes, we would like to have the capabilities of several monads at once
- examples
  - use several states; solution: combine all states into one record datatype
  - use writer and state; solution: use RWS monad
  - use state and error; solution: write dedicated monad (PGM parser monad)
- last example is tedious
- better solution: use **monad transformer**
  - monad transformer takes a monad as input, and then adds another effect
  - example: take `Maybe` as input monad, and then add capabilities of `State` on top of it
  - monad transformers are all indicated by suffix `T`
  - `newtype StateT s m a = ...`  
this is the monad transformer to add `State` features
  - `type State s = StateT s Identity`  
the `State` monad is just the `StateT` monad transformer where one plugs in the `Identity` monad
  - `newtype Identity a = Identity { runIdentity :: a }` is the trivial monad
- most monads that have been presented are part of **MTL**, the monad transformer library

## Review Definition of Known Monads Again

- most monads are actually defined via their corresponding monad transformers
- `data ParsecT s u m a = ...`  
`type Parsec s u = ParsecT s u Identity`
- `newtype RWST r w s m a = ...`  
`type RWS r w s = RWST r w s Identity`
- `newtype StateT s m a = ...`  
`type State s = StateT s Identity`
- ...; notable exception: for `IO` there is no `IOT` monad transformer
- with monad transformers we can easily combine multiple effects
  - `RWST r w s Maybe` combines `RWS` with `Maybe` error monad
  - `RWST r w s IO` combines `RWS` with `IO` monad
  - `StateT st (ParsecT s u m)` is monad transformer that adds `State` and `Parsec` features
  - `Identity` can always be used to terminate a stack of monad transformers, e.g.,  
`MT1 s (MT2 r (... MTn u Identity))`
- because of mentioned restriction, `IO` must always be at the inside

## Example: Just using IO Monad

- write function to list all subdirectories with number of entries per directory

```
listDirectory :: FilePath -> IO [String]
listDirectory d = filter notDots <$> getDirectoryContents d
  where notDots p = not $ p `elem` [".", ".."]

countEntries1 :: FilePath -> IO [(FilePath, Int)]
countEntries1 path = do
  contents <- listDirectory path
  rest <- flip mapM contents $ \name -> do
    let newName = path </> name
    isDir <- doesDirectoryExist newName
    if isDir
      then countEntries1 newName
      else return []
  return $ (path, length contents) : concat rest
```

## Example: Collect Output in Writer Monad via WriterT

```
countEntries2Main :: FilePath -> WriterT [(FilePath, Int)] IO ()
countEntries2Main path = do
  contents <- liftIO . listDirectory $ path
  tell [(path, length contents)]
  flip mapM_ contents $ \name -> do
    let newName = path </> name
    isDir <- liftIO . doesDirectoryExist $ newName
    when isDir $ countEntries2Main newName

countEntries2 :: FilePath -> IO [(FilePath, Int)]
countEntries2 = fmap snd . runWriterT . countEntries2Main
```

## Explanations

- `countEntriesMain :: ... -> WriterT [(FilePath, Int)] IO ()`
  - since the outer type is `WriterT`, the result type is an instance of `MonadWriter` [...]
  - therefore, `tell :: [...] -> WriterT [...] m ()` is available
- `liftIO :: MonadIO m => IO a -> m a` lifts IO-actions to a corresponding monad
  - `IO` is a trivial instance of `MonadIO` where `liftIO = id`
  - there also is an instance `(Monoid w, MonadIO m) => MonadIO (WriterT w m)`; this tells us that being an `MonadIO` instance is preserved by `WriterT w`
- `when :: Applicative f => Bool -> f () -> f ()` is if-then without else:  
`when p s = if p then s else pure ()`
- `runWriterT :: WriterT w m a -> m (a, w)`
  - run the `WriterT` monad transformer
  - result will be in original monad `m`
  - output of writer will be made available in second component of result
  - similar to `runWriter :: Writer w a -> (a, w)`
- overall: availability of both `MonadWriter` and `IO`;  
run `runWriterT` to convert `WriterT w m a` into `m (a,w)`, i.e., eliminate `WriterT`

RT (DCS @ UIBK)

Week 8

21/30

## Design of MTL

- several abstract classes, e.g., `MonadWriter`, `MonadReader`, `MonadState`, `MonadIO`,...
- several monad transformers, e.g., `WriterT`, `ReaderT`, `StateT`, ...
- $n \times n$  instance declarations
  - `(Monoid w, Monad m) => MonadWriter w (WriterT w m)` `MonadWriter` instance
  - `(Monoid w, MonadIO m) => MonadIO (WriterT w m)` `preserve MonadIO`
  - `(Monoid w, MonadState s m) => MonadState s (WriterT w m)` `preserve MonadState`
  - ...
  - `Monad m => MonadReader r (ReaderT r m)` `MonadReader` instance
  - `MonadIO m => MonadIO (ReaderT r m)` `preserve MonadIO`
  - `MonadState s m => MonadState s (ReaderT r m)` `preserve MonadState`
  - ...
- in total
  - allows flexible stacking of monad transformers:  
choose those transformers that are required for application
  - quite some effort to integrate new monad transformer:  
full implementation requires connection to all other abstract classes

RT (DCS @ UIBK)

Week 8

22/30

## Example for Stacking Monad Transformers

- example is an extension of the directory count example
  - extension 1: user must specify maximal recursion depth
  - extension 2: compute reached maximal recursion depth
- utilized monads
  - `MonadIO` is required for directory access
    - access via `liftIO :: MonadIO m => IO a -> m a`
  - use `MonadReader` to pass configuration around; that configuration stores recursion limit
    - access via `ask :: MonadReader r m => m r`
  - use `MonadState` to store the maximally reached recursion depth
    - access via `get :: MonadState s m => m s` and `put :: MonadState s m => s -> m ()`

RT (DCS @ UIBK)

Week 8

23/30

## Stacking of Monad Transformers Example – Setup

```
data AppConfig = AppConfig {  
    cfgMaxDepth :: Int  
} deriving (Show)
```

```
data AppState = AppState {  
    stDeepestReached :: Int  
} deriving (Show)
```

```
type App = ReaderT AppConfig (StateT AppState IO)
```

```
runApp :: App a -> Int -> IO (a, AppState)  
runApp app maxDepth =  
    let config = AppConfig maxDepth  
        state = AppState 0  
    in runStateT (runReaderT app config) state
```

RT (DCS @ UIBK)

Week 8

24/30

## Stacking of Monad Transformers Example – App

```
countEntries3Main :: Int -> FilePath -> App [(FilePath, Int)]
countEntries3Main curDepth path = do
  contents <- liftIO . listDirectory $ path
  allowedDepth <- cfgMaxDepth <$> ask
  rest <- flip mapM contents $ \name -> do
    let newPath = path </> name
    isDir <- liftIO $ doesDirectoryExist newPath
    if isDir && curDepth < allowedDepth
    then do
      let newDepth = curDepth + 1
      st <- get
      when (stDeepestReached st < newDepth) $
        put $ st { stDeepestReached = newDepth }
      countEntries3Main newDepth newPath
    else return []
  return $ (path, length contents) : concat rest
```

RT (DCS @ UIBK)

Week 8

25/30

## Final Steps

- wrapper for application that removes App type

```
countEntries3Main :: Int -> FilePath -> App [(FilePath, Int)]
runApp :: App a -> Int -> IO (a, AppState)

countEntries3 :: Int -> FilePath -> IO ([(FilePath, Int)], Int)
countEntries3 md fp =
  second stDeepestReached <$> runApp (countEntries3Main 0 fp) md
```

RT (DCS @ UIBK)

Week 8

26/30

## Limits of MTL

- when using MTL, one often can just use all features of the transformers in the stack
- there are two major exceptions
  - a single transformer occurs multiple times, e.g., `StateT Int (StateT String IO)`
    - what should be the type of `get`? return an `Int` or a `String`? how to access the other state?
  - monads outside MTL are used, where no automatic instance forwarding is available

```
class Monad m => MyMonad m where
  myFun :: Int -> a -> m [a]
```

```
foo :: MyMonad m => a -> ReaderT Int m a
foo x = do
  i <- ask
  {- how to invoke "xs <- myFun i x" at this point? -}
  return $ xs !! max i 5
```

- both problems can be solved by using `lift :: (MonadTrans t, Monad m) => m a -> t m a`
  - using `lift`, we get access to monad operations that are one level deeper in the stack
  - most (or even all) monad transformers in MTL instantiate `MonadTrans`

RT (DCS @ UIBK)

Week 8

27/30

`MonadTrans` and `lift :: (MonadTrans t, Monad m) => m a -> t m a`

- second problem solved

```
class Monad m => MyMonad m where myFun :: Int -> a -> m [a]
foo :: MyMonad m => a -> ReaderT Int m a
foo x = do
  i <- ask
  xs <- lift $ myFun i x
  return $ xs !! max i 5
```

  - here, `myFun i x :: m [a]`, so `lift $ myFun i x :: ReaderT Int m [a]`

- first problem solved

```
bar :: StateT Int (StateT String IO) ()
bar = do
  (x :: Int) <- read <$> liftIO getLine
  put x -- outer StateT
  (s :: String) <- lift $ get -- inner StateT
  liftIO $ putStrLn s
```

RT (DCS @ UIBK)

Week 8

28/30

## Design Decision

- in second problem from previous slide, one has two alternatives
- solution via `lift`
  - advantage: no instance declarations are required
  - disadvantage: application code needs to insert `lift`
- solution by writing instance declarations
  - disadvantage: a lot of boilerplate code has to be written ( $n \times n$  problem)
  - advantage: more comfort for the user – fewer manual liftings
- preferable solutions depends on number of required liftings

## Literature

- Real World Haskell, Chapters 16 and 18