Monad Transformers

Construção de compiladores I

Objetivos

Objetivos

- Apresentar o conceito de Monad Transformer para combinar diferentes efeitos colaterais em um programa.
- Uso de monad transformers para permitir a manipulação de diretórios no compilador de markdown.

Markdown

Markdown

- Iniciamos nosso curso com o projeto de um mini-compilador de Markdown.
- Objetivo: revisar conceitos da linguagem Haskell

Markdown

- Na última aula, finalizamos uma versão que processa apenas um único arquivo.
- Nesta aula estenderemos esse compilador com:
  - Processamento de diretórios
  - Incluir folhas de estilo nos html gerados.

Markdown

- Estender o compilador para processar diretórios é uma tarefa bem direta.

Markdown

- Incluir folhas de estilo
  - Tarefa simples, mas que leva a código longe do ideal.
  - Mostraremos como melhorar o código combinando mônadas.

Processando diretórios

Processando diretórios

- Para permitir processar todos os arquivos em um diretório, devemos modificar a interação por consoles.

Processando diretórios

- Modificação do tipo de opções de entrada.

```
data Options
  = Single Input Output
  | Directory FilePath FilePath
  deriving Show
```

Processando diretórios

- Modificação do parser

```
pOptions :: Parser Options
pOptions
  = subparser (singleParser <> directoryParser)
    where
      singleParser
        = command "file"
                  (info (helper <*> pSingle)
                        (progDesc "Convert a single markdown file to HTML"))
      directoryParser
        = command "directory"
                  (info (helper <*> pDirectory)
                        (progDesc "Convert all directory md files to HTML"))
```

Processando diretórios

- Parser de diretórios

```
pDirectory :: Parser Options
pDirectory =
  Directory <$> pInputDir <*> pOutputDir
```

Processando diretórios

- Parser de diretório de entrada

```
pInputDir :: Parser FilePath
pInputDir =
  strOption
    ( long "input"
      <> short 'i'
      <> metavar "DIRECTORY"
      <> help "Input directory"
    )
```

Processando diretórios

- Processamento de diretórios de saída é similar.

Processando diretórios

- Mudanças no pipeline de compilação.

```
startPipeline :: IO ()
startPipeline
  = do
      options <- optionsParser
      case options of
        Single inp out ->
          filePipeline False inp out
        Directory inpDir outDir ->
          directoryPipeline inpDir outDir
```

Processando diretórios

- Modificações no pipeline de arquivos individuais

```
filePipeline :: Bool -> Input -> Output -> IO ()
filePipeline dirMode inpFile outFile
  = do
      progressMessage dirMode inpFile
      (title,inpHandle,outHandle) <- createHandles inpFile outFile
      content <- hGetContents inpHandle
      res <- pipeline (title_ title) content
      hPutStrLn outHandle res
      hClose inpHandle
      hClose outHandle
```

Processando diretórios

- Pipeline de diretórios

```
directoryPipeline :: FilePath -> FilePath -> IO ()
directoryPipeline inputDir outputDir
  = do
      d <- directoryContents inputDir
      let files = filesToCompile d
          otherFiles = filesToCopy d
      let entries = map (\ (i,o) -> (FileInput i, FileOutput o)) files
      shouldContinue <- createOutputDirectory outputDir
      unless shouldContinue (hPutStrLn stderr "Cancelled." *> exitFailure)
      mapM_ (uncurry (filePipeline True)) entries
      let copy file = copyFile file (outputDir </> takeFileName file)
      mapM_ copy otherFiles
      putStrLn "Done."
```

Processando diretórios

- Código contendo exatamente essas modificações pode ser encontrado na branch `markup-directories`.

Adicionando estilos

Adicionando estilos

- Até o presente momento, não usamos folhas de estilo para formatar o HTML gerado.
- Como adicionar folhas de estilo?

Adicionando estilos

- Definido um tipo para armazenar o caminho de folhas de estilo

```
data Env
  = Env {
      stylePath :: FilePath
    } deriving Show

defaultEnv :: Env
defaultEnv = Env ""
```

Adicionando estilos

- Agora, temos que modificar todo o código para permitir este parâmetro adicional.

Adicionando estilos

- Adicionando tags para estilos na EDSL de HTML

```
stylesheet_ :: FilePath -> Head
stylesheet_ path =
    Head $ "<link rel=\"stylesheet\" type=\"text/css\" href=\"" <>
            escape path <> "\">"
```

Adicionando estilos

- Modificando o parser de diretórios para entrada

```
data Options
  = Single Input Output
  | Directory FilePath FilePath Env
  deriving Show
```

Adicionando estilos

- Modificando o parser de diretórios para entrada

```
pDirectory :: Parser Options
pDirectory =
  Directory <$> pInputDir <*> pOutputDir <*> pEnv
```

Adicionando estilos

- Modificando o parser de diretórios para entrada

```
pEnv :: Parser Env
pEnv = fromMaybe defaultEnv <$> optional p
  where
    p =  Env <$> strOption
                 (  long "style"
                 <> short 'S'
                 <> metavar "FILE"
                 <> help "Stylesheet filename"
                 )
```

Adicionando estilos

- Modificando o pipeline de compilação
  - Inclusão do environment em **TODAS** as funções do pipeline.

```
startPipeline :: IO ()
startPipeline
  = do
      options <- optionsParser
      case options of
        Single inp out ->
          filePipeline False defaultEnv inp out
        Directory inpDir outDir env ->
          directoryPipeline env inpDir outDir
```

Adicionando estilos

- Modificando o pipeline de compilação.
  - Estilos usados apenas em `filePipeline`.

  - Demais funções apenas "passam" o valor de `Env`.

```
filePipeline :: Bool -> Env -> Input -> Output -> IO ()
filePipeline dirMode env inpFile outFile
  = do
      progressMessage dirMode inpFile
      (title,inpHandle,outHandle) <- createHandles inpFile outFile
      content <- hGetContents inpHandle
      let header = title_ title <> stylesheet_ (stylePath env)
      res <- pipeline header content
      hPutStrLn outHandle res
      hClose inpHandle
      hClose outHandle
```

Adicionando estilos

- Versão contendo o código com passagem explícita de `Env` está disponível na branch `markup-explicit-env`.

Adicionando estilos

- Essa passagem de valores é tediosa e propensa a erros.
- Ideal: Acessar o valor somente no ponto onde este será utilizado.
  - Garantir que esse valor não será modificado.

Adicionando estilos

- Podemos garantir a situação ideal utilizando a mônada de somente leitura
  - Reader
- Como combinar essa mônada com a mônada de IO?

Adicionando estilos

- Para isso, devemos utilizar **transformadores monádicos**.
  - Permitem combinar a funcionalidade de diferentes mônadas.

Adicionando estilos

- Principal função da mônada de somente leitura
  - Retorna o valor armazenado na mônada para consulta.

```
ask :: MonadReader m => m a
```

Adicionando estilos

- Modificando o pipeline de compilação para usar a nova mônada

```
type CompilerM a = (ReaderT Env IO) a

runCompilerM :: Env -> CompilerM a -> IO a
runCompilerM env m = runReaderT m env
```

Adicionando estilos

- Modificando o início do pipeline

```
startPipeline :: IO ()
startPipeline
  = do
      options <- optionsParser
      case options of
        Single inp out ->
          runCompilerM defaultEnv (filePipeline False inp out)
        Directory inpDir outDir env ->
          runCompilerM env (directoryPipeline inpDir outDir)
```

Adicionando estilos

- Modificando o pipeline de arquivos individuais
  - Uso da função `ask`

```
filePipeline :: Bool -> Input -> Output -> CompilerM ()
filePipeline dirMode inpFile outFile
  = do
      env <- ask
      progressMessage dirMode inpFile
```

```
(title,inpHandle,outHandle) <- liftIO $ createHandles inpFile outFile
content <- liftIO $ hGetContents inpHandle
let header = title_ title <> stylesheet_ (stylePath env)
res <- pipeline header content
writeAndCloseHandles res inpHandle outHandle
```

Adicionando estilos

- Demais funções tiveram alterações pontuais
  - Modificação da assinatura de tipos
  - Uso de `liftIO`.

Adicionando estilos

- Código contendo essa versão está disponível na branch `main`.

Concluindo

Concluindo

- Com isso, terminamos nossa revisão de Haskell.
- Produzimos um compilador de um subconjunto de Markdown para HTML.
  - Utilizamos várias bibliotecas úteis de Haskell.
  - Apresentamos o padrão functional core / imperative shell

Concluindo

- Utilizamos monad transformers em um exemplo muito simples.
- Recomendo ver um exemplo um pouco mais interessante disponível no seguinte repositório.

Concluindo

- Próxima aula: iniciamos o primeiro tópico de compiladores
  - Análise léxica

Exercícios

Exercícios

- Em exercícios anteriores, você extendeu o compilador de Markdown para gerar slides beamer.
  - Modifique a versão atual para gerar slides para todos os arquivos em um diretório

Exercícios

- Modifique o parser de argumentos de linha de comando para permitir a especificação do template de slides a ser usado como argumento de linha de comando.
  - Lista de templates disponíveis.
- Entrega via Github classroom.