Yet Another Monad Tutorial

Parallel Functional Programming (DAT280)

John Hughes

Recall Input/Output in Haskell...

```
writeFile ”baz” s  :: IO ()
```

Write s :: String to file baz

```
readFile ”foo”    :: IO String
```

”An IO action returning a String”

Read file foo and return its contents

”How do I get the String out of an IO String?”

```
writeFile "baz" (readFile "foo")
```

```
Couldn't match type ‘IO String’ with ‘[Char]’
     Expected type: String
       Actual type: IO String
    In the second argument of ‘writeFile’, namely ‘(readFile "foo")’
```

Recall Input/Output in Haskell…

```
writeFile ”baz” s  :: IO ()
```

Write s :: String to file baz

```
readFile ”foo” s  :: IO String
```

”An IO action returning a String”

Read file foo and return its contents

```
      String                          IO String

         do s <- readFile "foo"
            writeFile "baz" s      :: IO ()
```

Well typed

Everything that does I/O has an IO type

With great power...

Mutable variable containing an a

```
import Data.IORef

      newIORef   :: a -> IO (IORef a)
      readIORef  :: IORef a -> IO a
      writeIORef :: IORef a -> a -> IO ()
```

```
increment :: IORef Integer -> IO Integer

increment r = do n <- readIORef r
                 writeIORef r (n+1)
                 return n
```

r++

…comes great responsibility

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

```
bad = do r <- newIORef 0
         return (increment r - increment r)
```

+1 or -1?

```
No instance for (Num (IO Integer)) arising from a use of ‘-’
    In the first argument of ‘return’, namely
      ‘(increment r - increment r)’
```

…comes great responsibility

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

```
bad = do r <- newIORef 0
         a <- increment r
         b <- increment r
         return (a - b)
```

-1 **+1**

Haskell forces you to *sequence* all input/output

Compare…

```
twice io = do a <- io
              b <- io
              return (a,b)
```

```
twoOf io = do a <- io
              return (a,a)
```

```
IO a -> IO (a,a)
```

```
*Ex> r <- newIORef 0

*Ex> twoOf (increment r)
(0,0)

*Ex> twice (increment r)
(1,2)
```

?


WAIT!
WHAT IS GOING ON?!
quickmeme.com

Overloading in Haskell

```
*Ex> 1 + 2                :: Integer
3
*Ex> 1.0 + 2.0            :: Double
3.0
*Ex> :t (+)
(+) ::          a -> a -> a
*Ex> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  …
instance Num Integer -- Defined in 'GHC.Num'
instance Num Double -- Defined in 'GHC.Float'
…
```

Writing your own instances

```
data Complex a = a :+ a
```

e.g. Complex Float,
Complex Double…

```
instance Num a => Num (Complex a) where
   (x:+y) + (x':+y') = (x+x') :+ (y+y')
   (x:+y) - (x':+y') = (x-x') :+ (y-y')
   (x:+y) * (x':+y') = (x*x'-y*y') :+ (x*y'+y*x')
   …
```

A different implementation of
(+) etc. for each type

```
sumsq x y = x*x + y*y
```

```
sumsq :: Num a => a -> a -> a
```

Handling failure in Haskell

```
*Ex> 5 `div` 2
2
*Ex> 5 `div` 0
*** Exception: divide by zero
```

```
data Maybe a =
    Nothing
  | Just a
```

```
safeDiv m n =
  if n==0 then            else          m `div` n
```

```
*Ex> 5 `safeDiv` 2
Just 2
*Ex> 5 `safeDiv` 0
Nothing
```

`1 + (a `safeDiv` b)`

```
Couldn't match expected type ‘Integer’
                 with actual type ‘Maybe Integer’
    In the expression: 1 + (a `safeDiv` b)
```

”How do I get the Integer out of a Maybe Integer?”

```
import Data.Maybe

fromJust (Just a) = a
```

```
1 + fromJust (a `safeDiv` b) :: Maybe Integer
```

```
*Ex> 1 + fromJust (5 `safeDiv` 0)
*** Exception: Maybe.fromJust: Nothing
```

```
fmap (    )    1 + (a `safeDiv` b)      Maybe
                                       Integer
    Integer                  Maybe
      ->                    Integer
    Integer
```

```
fmap :: (a->b) -> Maybe a -> Maybe b
fmap f (Just n) = Just (f n)
fmap f Nothing  = Nothing

map  :: (a->b) -> [a] -> [b]

fmap :: (a->b) -> IO a -> IO b
fmap f ioA = do a <- ioA
                return (f a)
```

Overloaded fmap

```
*Ex> fmap (+1) (Just 3)
Just 4

*Ex> fmap (+1) [1,2,3]
[2,3,4]

*Ex> fmap (length . lines) (readFile "Ex.hs")
37
```

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor Maybe …
instance Functor [] …
instance Functor IO …
```

```
fmap ( ) (a `safeDiv` b) <*> (c `safeDiv` d)
```

Integer -> Integer -> Integer

Maybe Integer

Maybe Integer

Maybe (Integer -> Integer)

```
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
Just f <*> Just a = Just (f a)
_      <*> _      = Nothing
```

We don’t need **`fmap`** any more!

```
fmap f maybeA =  pure f <*> maybeA
```

```
pure (+) <*> (a `safeDiv` b)
         <*> (c `safeDiv` d)
```

```
pure = Just
```

We can do the same for lists and IO!

```
(<*>) :: [a -> b] -> [a] -> [b]

fs <*> as = [f a | f <- fs, a <- as]
pure a = [a]

(<*>) :: IO (a -> b) -> IO a -> IO b

ioF <*> ioA = do f <- ioF
                 a <- ioA
                 return (f a)
pure a = return a
```

Applicative Functors

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

instance Applicative Maybe …
instance Applicative [] …
instance Applicative IO …
```

```
(a `safeDiv` b) `safe(Div` c      )
```

```
 Maybe                              Integer
Integer                               ->
                                     Maybe
                                    Integer
```

fmap

```
Maybe
(Maybe
Integer)
```

```
(>>=) :: Maybe Integer ->
         (Integer -> Maybe Integer) -> Maybe Integer
```

>>=

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

Just a  >>= f = f a
Nothing >>= _ = Nothing


(>>=) :: [a] -> (a -> [b]) -> [b]

as >>= f = [b | a <- as, b <- f a]


(>>=) :: IO a -> (a -> IO b) -> IO b

ioA >>= f = do a <- ioA
               f a
```

Monads

```
class Applicative m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

”bind”

```
instance Monad Maybe …
instance Monad [] …
instance Monad IO …
```

We don’t need `(<*>)` any more!

```
mf <*> ma =  mf >>= (\f -> ma >>= (\a -> return (f a)))
```

m (a->b)   m a   m (a->b)   a->b   m a   a   m b   b

The Meaning of **do**

```
do a <- ma
   body…
```
⇒
```
ma >>= (\a -> do body…)
```

```
do ma
   body…
```
⇒
```
do _ <- ma
   body…
```

```
do ma
```
⇒
```
ma
```

```
do s <- readFile "foo"
   writeFile "baz" s
```
⇒
```
readFile "foo" >>= (\s ->
    do writeFile "baz" s)
```
⇒
```
readFile "foo" >>= (\s ->
       writeFile "baz" s)
```

What have we seen so far?

`Maybe a, [a], IO a` — all ”ways of producing” **a**s

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

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

class Applicative m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

do-notation

Libraries that work with any **`Monad`**/**`Applicative`**/**`Functor`**

Designing a parsing library

What should the type of a parser be?

```
type Parser a = String -> a
```

Suppose we have **`number :: Parser Integer`**

```
*Ex> number "123"
123

*Ex> number "123,456"
*** Exception: Ex.hs:79:1-53: Non-exhaustive patterns
in function number
```

Designing a parsing library

What should the type of a parser be?

```
type Parser a = String -> Maybe
```

Suppose we have **`number :: Parser Integer`**

`*Ex> number "123"`
`Just 123`

`*Ex> number "123,456"`
~~`Nothing`~~

~~`Just 123`~~

`Just (123, ",456")`

How can I *reuse* **`number`** to parse a sequence of numbers?

Designing a parsing library

What should the type of a parser be?

```
type Parser a = String -> Maybe (a, String)
```

Suppose we have `number :: Parser Integer`

```
*Ex> number "123"
Just (123,"")

*Ex> number "123,456"
Just (123,",456")

*Ex> number "-123"
Nothing
```

The type defines a *calling convention* for parsing functions

Designing a parsing library

What should the type of a parser be?

```
type Parser a = String -> Maybe (a, String)
```

Suppose we have **`number :: Parser Integer`**

The type defines a *calling convention* for parsing functions

```
*Ex> number "123"
Just (123,"")

*Ex> number "123,456"
Just (123,",456")

*Ex> number "-123"
Nothing
```

```
newtype            = Parser (                              )
```

Example: a basic parser

```
newtype Parser a = Parser (String -> Maybe (a,String))

literal :: Char -> Parser Char

literal c = Parser $
  case s of
    x:xs | x==c -> Just (x,xs)
    _           -> Nothing

unParser (Parser p) = p
```

```
*Ex> unParser (literal 'a') "abc"
Just ('a',"bc")

*Ex> unParser (literal 'a') "bc"
Nothing
```

The Parser Monad

```
instance Monad Parser where
  return a = Parser $ \s -> Just (a,s)
```

Parser a   a -> Parser b

(b,String)

```
  Parser p >>= f = Parser $ \s ->
    p s >>= \(a,s') -> let Parser p' = f a in p' s’
```

Maybe
(a,String)

Parser b

```
instance Applicative Parser where
  pure = return
  (<*>) = ap
```

import Control.Monad

```
instance Functor Parser where
  fmap f x = pure f <*> x
```

"123,456"

```
twoNumbers = do m <- number
                literal ','
                n <- number
                return (m,n)
```

```
*Ex> unParser twoNumbers "123,456"
Just ((123,456),"")
```

```
twoNumbers = (,) <$> number <* literal ',' <*> number
```

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

Parsing a digit

```
satisfies :: (Char->Bool) -> Parser Char

satisfies p = Parser $ \s ->
  case s of
    x:xs |  p x -> Just (x,xs)
    _           -> Nothing
```

```
*Ex> unParser (satisfies isDigit) "123"
Just ('1',"23")

*Ex> unParser (satisfies isDigit) "456"
Just ('4',"56")
```

```
import Data.Char
```

Choosing between alternatives

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

```
instance Alternative Maybe where
  empty = Nothing

  Just a  <|> _ = Just a
  Nothing <|> b = b
```

```
instance Alternative [] where
  empty = []
  xs <|> ys = xs ++ ys
```

```
instance Alternative IO ???
```

Choosing between parsers

```
instance Alternative Parser where
  empty = Parser $ \s -> empty
  Parser f <|> Parser g = Parser $ \s -> f s <|> g s
```

```
many, some :: Alternative f => f a -> f [a]

many p = some p <|> pure []

some p = (:) <$> p <*> many p
```

Parsing numbers

```
*Ex> unParser (satisfies isDigit) "123,456"
Just ('1',"23,456")

*Ex> unParser (some (satisfies isDigit)) "123,456"
Just ("123",",456")

*Ex> unParser (read <$> some (satisfies isDigit) :: Parser Integer)
"123,456"
Just (123,",456")
```

```
number :: Parser Integer
number = read <$> some (satisfies isDigit)
```

Parsing a list of numbers

```
listOfNumbers :: Parser [Integer]

                            (*>) :: f a -> f b -> f b
listOfNumbers =
  (:) <$> number <*> many (literal ',' *> number)


*Ex> unParser listOfNumbers "12,34,5"
Just ([12,34,5],"")


listOfNumbers = commaSeparated number

commaSeparated p =
      (:) <$> p <*> many (literal ',' *> p)
  <|> pure []
```

Monad transformers: libraries to build monads

```
newtype Parser a = Parser (StateT String Maybe a)
```

```
newtype StateT s m a = StateT (s -> m (a, s))
```

Control.Monad.State

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
```

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving,
             FlexibleContexts #-}

module PL where

import Control.Applicative
import Control.Monad.State

newtype Parser a = Parser (StateT String Maybe a)
  deriving (Functor,Applicative,Alternative,Monad,
            MonadState String)

runParser (Parser p) = runStateT p

satisfies p = do
  s <- get
  case s of
    x:xs | p x -> do put xs
                     return x
    _          -> empty

literal c = satisfies (==c)
```

A Simple Calculator

```
*Calc> runParser expr "1+2+3"
Just (6.0,"")

*Calc> runParser expr "1-2-3"
Just (-4.0,"")

*Calc> runParser expr "1+2*3+4*5"
Just (27.0,"")

*Calc> runParser expr "(1+2)/(3+4)"
Just (0.4285714285714285 5,"")
```

1 (- 2) (- 3)

( )

```
module Calc where

import Data.Char
import Control.Applicative

import PL

expr :: Parser Double
expr = leftAssoc term (plusMinus <*> term)
term = leftAssoc factor (timesDiv <*> factor)
factor = number
     <|> literal '(' *> expr <* literal ')'

number = read <$> some (satisfies isDigit)

plusMinus = flip <$> ((+) <$ literal '+' <|> (-) <$ literal '-')
timesDiv  = flip <$> ((*) <$ literal '*' <|> (/) <$ literal '/')

leftAssoc pa pb = do
  a <- pa
  bs <- many pb
  return (foldl (flip id) a bs)
```

expr ::= term ((+|-) term)*
term ::= factor ((*|/) factor)*
factor ::= digit+ | ’(’ expr ’)’

Not a complete parsing library…

- White space
- Separate lexical analysis
- Error messages and locations
- Layout-sensitive parsing
- Operator precedences
- …

Comprehending Monads

Philip Wadler
University of Glasgow*

d *monads* in the 1960's to
spects of universal algebra.
vented *list comprehensions*
xpress certain programs in-

ture by straightforward th
stance, a counter can be sir
evant functions to accept a
counter
(the c

Lisp & FP'90
1340

Imperative functional programming

Simon L Peyton Jones Philip Wadler

Dept of Computing Science, University of Glasgow
Email: {simonpj,wadler}@dcs.glasgow.ac.uk

model, based on monads, for perform-
in a non-strict, purely functional lan-
osable, extensible, efficient, requires no

grams that do so (Section 2). Comb
order fu
highly
perform

POPL'93
783

FUNCTIONAL PEARL

Applicative programming with effects

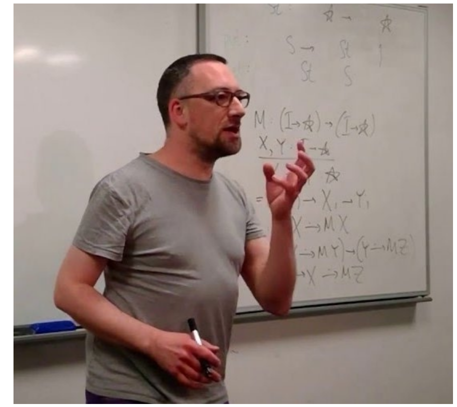
CONOR MCBRIDE[1]
University of Nottingham

ROSS PATERSON
City University, London

Abstract

In this article, we introduce Applicative functors – an abstract characterisation of an applicative style of effectful programming, weaker than Monads ar

JFP'08
565



FUNCTIONAL PEARL

Monadic parsing in Haskell

GRAHAM HUTTON
University of Nottingham, Nottingham, UK

ERIK MEIJER
University of Utrecht, Utrecht, The Netherlands

1 Introduction

This paper is a tutorial on defining recursive descent parsers in Haskell. In the spirit of *one-stop shopping*, the paper combines material from three areas into a single source. The three areas are functional parsers (Burge, 1

JFP'98
247