Introduction to Functional Programming
Part 4: Monads and Applicative Functors

Jeremy Gibbons
University of Oxford

4.1 Separation of Church and state

- a pure functional language such as Haskell is *referentially transparent*
- expressions do not have side-effects
- the sole purpose of an expression is to denote a value
- but what about computations that perform I/O
  (eg printing to the console or writing to the file system?)
- how to incorporate these into Haskell?

4.1 Monadic I/O

- introduce a new type of I/O computations
- *IO a* is type of computation that may do I/O, then returns an element of type *a*
- *idea:* *putStr* `"foo"` has *no* effect at all

$$putStr :: String \to IO\ ()$$

- *IO a* can be seen as the type of a *to-do list*
- to-do list *vs* actually doing something
- specifying vs executing an I/O computation
- *main* has type *IO* ()
- *only* the to-do list bound to *main* is executed

4.2 Discovering monads

- *IO* is a *monad*
- monads form *an abstract datatype of computations*
- computations in general may have *effects*:
  I/O, exceptions, mutable state, etc
- monads are a mechanism for cleanly incorporating such
  impure features in a *pure* setting
- other monads encapsulate exceptions, state, non-determinism, etc

- the following slides motivate the need for a general notion of computation

4.2 An evaluator

Here's a simple datatype of terms:

```
data Expr = Lit Integer | Div Expr Expr
  deriving (Show)

good, bad :: Expr
good = Div (Lit 7) (Div (Lit 4) (Lit 2))
bad  = Div (Lit 7) (Div (Lit 2) (Lit 4))
```

. . . and an evaluation function:

```
eval :: Expr → Integer
eval (Lit n)     = n
eval (Div d e) = eval d `div` eval e
```

4.2 Exceptions

Evaluation may fail, because of division by zero.
Let's handle the exceptional behaviour:

```
evalE :: Expr → Maybe Integer
evalE (Lit n)     = Just n
evalE (Div d e) =
  case evalE d of
    Nothing → Nothing
    Just m   → case evalE e of
                 Nothing → Nothing
                 Just n    →
                   if n == 0 then Nothing
                             else Just (m `div` n)
```

4.2 Counting

We could instrument the evaluator to count evaluation steps:

```
type Counter a = State → (a, State)
type State = Int

evalC :: Expr → Counter Integer
evalC (Lit n)     = λi → (n, i + 1)
evalC (Div d e) = λi → let (m, i') = evalC d (i + 1)
                           (n, i'') = evalC e i'
                       in (m `div` n, i'')
```

4.2 Tracing

. . . or to trace the evaluation steps:

```
type Trace a = (Output, a)
type Output = String

evalT :: Expr → Trace Integer
evalT (Lit n)     = (line (Lit n) n, n)
evalT (Div d e) = let
                      (s, m) = evalT d
                      (s', n) = evalT e
                      p = m `div` n
                  in (s ++ s' ++ line (Div d e) p, p)

line :: Expr → Integer → Output
line t n = "   " ++ show t ++ " yields " ++ show n ++ "\n"
```

4.2 Ugly!

- none of these extensions is difficult
- but each is rather awkward
- each obscures the previously clear structure

- how can we simplify the presentation?
- what do they have in common?

4.3 The monad type class

The commonalities can be captured by a type class:

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

(plus some laws).
Haskell also provides **do**-notation for *Monad* instances.

4.3 Original evaluator, monadically

$$
\begin{aligned}
&evalM :: (Monad\ m) \Rightarrow Expr \to m\ Integer \\
&evalM\ (Lit\ n) \quad\ = return\ n \\
&evalM\ (Div\ d\ e) = evalM\ d \gg\!= \lambda m \to \\
&\qquad\qquad\quad\ \ evalM\ e \gg\!= \lambda n \to \\
&\qquad\qquad\quad\ \ return\ (m\ `div`\ n)
\end{aligned}
$$

Still pure, but written in the monadic style; much easier to extend.

4.3 Original evaluator, using do notation

```
evalM :: (Monad m) ⇒ Expr → m Integer
evalM (Lit n)     = do return n
evalM (Div d e) = do m ← evalM d
                     n ← evalM e
                     return (m `div` n)
```

do-notation is defined in terms of $\gg=$.

4.3 The exception instance

Exceptions instantiate the class:

```
instance Monad Maybe where
  return a       = Just a
  Nothing >>= _  = Nothing
  Just a  >>= f  = f a
```

The effect-specific behaviour is to throw an exception:

```
throw :: Maybe a
throw = Nothing
```

4.3 Exceptional evaluator, monadically

```
evalE :: Expr → Maybe Integer
evalE (Lit n)     = do return n
evalE (Div d e) = do m ← evalE d
                     n ← evalE e
                     if n == 0 then throw
                               else return (m `div` n)
```

4.3 The counter instance

Counters instantiate the class.

```
type Counter a = State → (a, State)
instance Monad Counter where
  return a = λi → (a, i)
  ma >>= f = λi → let (a, i') = ma i in f a i'
```

The effect-specific behaviour is to increment the count:

```
tick :: Counter ()
tick = λi → ((), i + 1))
```

(In fact, type class instances must be **data**, not plain **type**s.)

4.3 Counting evaluator, monadically

```
evalC :: Expr → Counter Integer
evalC (Lit n)     = do tick
                       return n
evalC (Div d e) = do tick
                       m ← evalC d
                       n  ← evalC e
                       return (m `div` n)
```

4.3 The tracing instance

Tracers instantiate the class:

```
type Trace a = (Output, a)
instance Monad Trace where
  return a    = ("", a)
  (s, a) >>= f = let (s', b) = f a in (s ++ s', b)
```

The effect-specific behaviour is to log some output:

```
trace :: String → Trace ()
trace s = (s, ())
```

4.3 Tracing evaluator, monadically

```
evalT :: Expr → Trace Integer
evalT (Lit n)     = do trace (line (Lit n) n)
                       return n
evalT (Div d e) = do m ← evalT d
                     n  ← evalT e
                     let p = m `div` n
                     trace (line (Div d e) p)
                     return p
```

4.4 The IO monad

There's *no magic* to monads in general: all the monads above are just plain (perhaps higher-order) data, implementing a particular interface.

But there is one magic monad: the *IO* monad. Its implementation is abstract, hard-wired in the language implementation.

```
data IO a = ...
instance Monad IO where ...
```

4.4 Interpreting strings

- if evaluating at non-*IO* type, prints value; for *IO*, performs computation
- strings as values get displayed as strings:

```
? "Hello,\nWorld"
"Hello,\nWorld"
```

- *putStr* turns a string into an outputting computation:

```
? putStr "Hello,\nWorld"
Hello,
World
```

4.4 Example

A simple interactive program:

```
welcome :: IO ()
welcome
  = putStr "Please enter your name.\n" >>=
    getLine >>= λs →
    putStr ("Welcome " ++ s ++ "!\n")
```

The operator $\gg$ is a convenient shorthand:

```
(>>)  :: (Monad m) ⇒ m a → m b → m b
m >> n = m >>= λ_ → n
```

4.4 IO computations as first-class citizens

- we can freely mix IO computations with, say, lists

$$
\begin{aligned}
&\mathit{main} :: \mathit{IO}\ () \\
&\mathit{main} = \mathit{sequence}\ [\mathit{putStrLn}\ (\mathit{show}\ i) \mid i \leftarrow [0\,..\,9]]
\end{aligned}
$$

- don't forget the list pattern

$$
\begin{aligned}
&\mathit{sequence} :: [\mathit{IO}\ ()] \to \mathit{IO}\ () \\
&\mathit{sequence}\ [\,] \qquad = \mathit{return}\ () \\
&\mathit{sequence}\ (a : \mathit{as}) = a \gg \mathit{sequence}\ \mathit{as}
\end{aligned}
$$

  (the predefined version of *sequence* is more general)
- IO computations are first-class citizens!
- Haskell is the world's finest imperative language!

4.4 More IO operations

```
print    :: (Show a) ⇒ a → IO ()
readLn   :: (Read a) ⇒ IO a

putChar  :: Char → IO ()
getChar  :: IO Char

type FilePath = String
writeFile :: FilePath → String → IO ()
readFile  :: FilePath → IO String

data StdGen = ...              -- standard random generator
class Random where ...         -- randomly generatable
randomR :: (Random a) ⇒ (a, a) → StdGen → (a, StdGen)
getStdRandom :: (StdGen → (a, StdGen)) → IO a
```

and many more ...

4.4 Examples: character I/O

```
putStr, putStrLn :: String → IO ()
putStr ""     = do return ()
putStr (c : s) = do putChar c
                    putStr s
putStrLn s    = do putStr s
                    putChar '\n'

getLine :: IO String
getLine = do c ← getChar
             if c == '\n' then return ""
                          else do s ← getLine
                                  return (c : s)
```

4.4 File I/O

```
processFile :: FilePath → (String → String) → FilePath → IO ()
processFile inFile f outFile
  = do s ← readFile inFile
       let s' = f s
       writeFile outFile s'
```

In general, try to minimize the I/O part of your program.

4.4 Random numbers

```
import System.Random

rollDie :: IO Int
rollDie = getStdRandom (randomR (1,6))

roll3Dice :: IO Int
roll3Dice = do x ← rollDie
               y ← rollDie
               z ← rollDie
               return (x + y + z)
```

4.5 Applicative functors

- monads are a very *natural* abstraction
- defined in maths long before their use in computing

- but they are also a very *powerful* abstraction
- convenient to use—but harder to define, and to analyse

- sometimes a *weaker* abstraction is sufficient
- . . . which is then easier to define and to analyse
- *applicative functors* are one such

4.5 Definition of applicatives

```
class Applicative m where
  pure :: a → m a
  (⊛)  :: m (a → b) → m a → m b   -- written "<*>" in ASCII
```

(with some laws).

Compare with monads:

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

4.5 Every monad is an applicative...

When m is a monad, one can define

$$\begin{aligned} \mathit{pure}\ a &= \mathit{return}\ a \\ mf \circledast ma &= \mathbf{do}\ \{f \leftarrow mf; a \leftarrow ma; \mathit{return}\ (f\ a)\} \end{aligned}$$

Indeed, the Haskell libraries (now) enforce this:

$$\mathbf{class}\ \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m\ \mathbf{where}\ \ldots$$

(the direction of $\Rightarrow$ is awkward here!).

4.5 ... but some applicatives are not monads

For example:

```
data Const a b = C a
instance Monoid a ⇒ Applicative (Const a) where
  pure _     = C mempty              -- mempty is the unit of the monoid
  C x ⊛ C y = C (mappend x y)       -- mappend is the multiplication of the monoid
```

So applicatives are more general than monads.

4.5 Static analysis

Compare types of applicative and monadic "sequential composition":

$$(\circledast) \quad :: m\,(a \to b) \to m\,a \to m\,b$$
$$(\gg\!=) :: m\,a \to (a \to m\,b) \to m\,b$$

$\gg\!=$ provides the power to choose second computation based on result of first.
But then you can't analyse the second until you have executed the first!

$\circledast$ is less powerful, and therefore more amenable.
Used eg for parser combinators, for remote execution...

4.5 Traversal

Don't need full power of monads for *effectful traversal*:

```
class Traversable t where
  traverse :: Applicative m ⇒ (a → m b) → t a → m (t b)
```

(again, with some laws). For example, left-to-right traversal of lists:

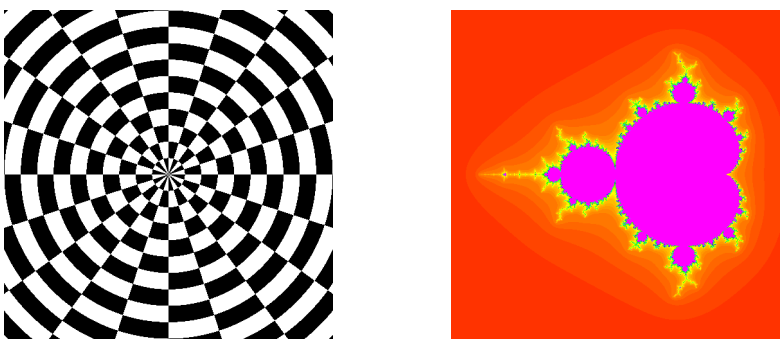
```
instance Traversable [ ] where
  traverse f [ ]       = pure [ ]
  traverse f (x : xs) = pure (:) ⊛ f x ⊛ traverse f xs
```

so

```
Main⟩ let body s = do { putStrLn s; return (length s) }
Main⟩ traverse body ["abc","de","fghi"]
abc
de
fghi
[3,2,4]
```

A Bitmap images

This is a series of exercises on generating and rendering images. We start off with rendering simple bitmap images. Later, we will look at generating those images—in particular, in Section A.6 we will generate some 'Op art' images such as the polar chequerboard in Figure 3(a) below, and in Section A.7 some fractal images such as the *Mandelbrot Set* shown in Figure 3(b).



(a) (b)

Figure 3: (a) Some Op Art, and (b) a part of the Mandelbrot Set