Chapter 7: Monads

Learning targets of this chapter:

1. Reduce reservations against monads
2. Monads as Haskell class of data types
3. Types and basic functions
4. Syntax: `do`, `let`, blocks and scopes
5. Referential transparency of monads
6. Programming of input/output operations
7. Combinators on monads
8. Applications: state monad, monadic parsing
9. Combination of several monads

Reservations Against Monads

Often, the concept of a monad triggers certain associations

- „Monads are for input/output.“
- „Monads are for side-effects or states.“
- „You can never exit a monad.“
- „Monads are a crutch to enable imperative programming in Haskell.“
- „Monads are weird abstractions from category theory.“

Monads are much more than just a vehicle for computations with side-effects.

The power of monads is rooted in their individual composition operator, not in some side-effects.

A Very Simple Tree Data Type

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)


unit :: a -> Tree a
unit x = Leaf x

tree01 :: Tree Int
tree01 = Node (Leaf 0) (Leaf 1)
```

A Simple Function on `Tree a`

```
inc :: Num a => Tree a -> Tree a
inc (Leaf x)   = Leaf (x+1)
inc (Node s t) = Node (inc s) (inc t)
```

With abstraction:

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x)   = Leaf (f x)
mapTree f (Node s t) = Node (mapTree f s) (mapTree f t)
```

```
inc :: Num a => Tree a -> Tree a
inc t = mapTree (+1) t
```


Type Class `Functor`

The following pattern:

> Apply function `f :: a -> b` to each value of type `a` that occurs „in“ a data object with values of type `a`.

is condensed in type class `Functor`.

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

instance Functor Tree where
    fmap = mapTree
```

Here, `f` is a variable name for a type *constructor*.

Continuing with `Tree`

Obviously, the following function `rightInc` cannot be defined with `mapTree`:

```
rightInc :: Num a => Tree a -> Tree a
rightInc (Leaf x)   = Node (Leaf x) (Leaf (x+1))
rightInc (Node s t) = Node (rightInc s) (rightInc t)
```

However, `rightInc` can be expressed with `mapTree` and `joinTree`:

```
rightIncLeaf :: Num a => a -> Tree a
rightIncLeaf x = Node (Leaf x) (Leaf (x+1))

joinTree :: Tree (Tree a) -> Tree a
joinTree (Leaf t)   = t
joinTree (Node s t) = Node (joinTree s) (joinTree t)

flatMapTree :: (a -> Tree b) -> Tree a -> Tree b
flatMapTree f t = joinTree (mapTree f t)


rightInc t = flatMapTree rightIncLeaf t
```


Type Class Monad

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

instance Monad Tree where
    -- unit :: a -> Tree a
    return x  =  unit x

    -- flatMapTree :: (a -> Tree b) -> Tree a -> Tree b
    t >>= f   =  flatMapTree f t
```

This definition „just“ introduces new names:

- `return` or `unit` generates a leaf,
- `(>>=)` or `flatMapTree` extends a tree at every leaf by an (in dependence of the according leaf) generated subtree.

Interpretation

Analogously to `fmap` in type class `Functor`:

(„Apply `f :: a -> b` to each value of type `a`
that occurs „in“ a data object with values of type `a`.“),

one can interpret `(>>=)` in type class `Monad`:

„(With help of a function) For each value of type `a`, generate in a data object a new (sub)object and *attach* it at the corresponding place.“

(Depending on the monad, *attach* can have widely varying meanings.)

Basic Definitions

- Type class `Monad` for monads with operations `return` and `(>>=)` (bind)
- Generation of a monad:

  ```
  return :: Monad m => a -> m a
  ```

  – takes a value x

  – returns a monadic operation that returns only x
- Composition of two monadic computations:

  ```
  (>>=) :: Monad m => m a -> (a -> m b) -> m b
  ```

  – the second computation may use the return value of the first

  – the overall return value is that of the second computation

Note: In newer Haskell standards, `Functor` and `Applicative` are superclasses of class `Monad`; therefore, one needs to define instances for these classes, too.

Class `Applicative`

The class `Applicative` captures structures that are between `Functor` and `Monad`, i.e., they are more restricted than proper monads.

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

- `pure` plays the same role as `return` in monads,
- `(<*>)` expresses application of a function (wrapped in the applicative structure) to a value (also wrapped in the applicative structure).

Template for Defining Instances

When a data type `T` is a monad, one can use the following template to define instances for `Functor`, `Applicative` and `Monad`:

```
import Control.Monad (ap)
instance Functor T where
    fmap f x = x >>= pure . f
instance Applicative T where
    pure   = ...   -- the monad's return operation
    (<*>)  = ap
instance Monad T where
    (>>=)  = ...   -- the monad's bind operation
```

Note that the definition of `return` is actually in the definition of `pure` – this is because `return` today has `pure` as its default implementatian; in the future, it may be removed from class `Monad` and become a free function aliasing `pure`.

Application of Instance Monad Tree

With the `Monad` instance of `Tree`, we can specify functions, that can be expressed via `unit` and `flatMapTree`, differently:

```
rightInc t  =  flatMapTree rightIncLeaf t

rightInc t  =  t >>= rightIncLeaf

-- do Notation (see the following slides)

rightInc t  =  do { x <- t; rightIncLeaf x }

rightInc t  =  do { x <- t; y <- rightIncLeaf x; return y }
```

do-Notation

The `do`-notation is just syntactic sugar for `return` and `(>>=)`, but appears „imperative“.

```
do { stmt }                 =  stmt
do { x <- stmt; stmts }     =  stmt >>= (\x -> do { stmts })
do { stmt; stmts }          =  stmt >>= (\_ -> do { stmts })
do { let y = x; stmts }     =  let y = x in do { stmts }
```

Example: `do { x <- t; rightIncLeaf x }`

$\leadsto$ `t >>= (\x -> do { rightIncLeaf x })`

$\leadsto$ `t >>= (\x -> rightIncLeaf x)`

(and with η-conversion)

$\leadsto$ `t >>= rightIncLeaf`

do-Notation and Layout Style

```
foo a = return a >>= (\b -> (f b >>= (\c -> g b c)))
```

In do-notation:

```
foo a = do { b <- return a; c <- f b; g b c }
```

In layout style:

```
foo a = do
  b <- return a
  c <- f b
  g b c
```

If also value c should be returned:

```
foo b = do
  c <- f b
  d <- g b c
  return (c,d)
```

Scopes of Variables

Each binding with `<-` in `do`-notation opens a new scope.

Local bindings with `let` are recursive in `do`-blocks.

Example:

```
foo r = do
         x <- f r
         x <- g x          -- new variable x
         let ys = x : ys
         x <- h x ys       -- new variable x
         return x
```

because of its correspondence with:

```
foo r = f r >>= (\x ->
        g x >>= (\x ->
        let ys = x : ys in h x ys >>= (\x ->
        return x)))
```

Monadic Re-Assignment

```
do x <- return 1
   let y = x
   x <- return 2
   putStrLn $ show (y,x,y == x)
```

Each binding of `x` with `<-` introduces a new variable; it is just syntactic sugar for λ-abstraction.

Monad Laws

The requirements on *unit* and *join* carry over to `return` and `(>>=)` as follows:

```
return a >>= k             =  k a
m >>= return               =  m
m >>= (\x -> k x >>= h)    =  (m >>= k) >>= h
```

and (this can be used as a definition for `fmap` if no `Functor` instance is defined):

```
fmap f xs                  =  xs >>= return . f
```

These requirements need to be verified for every monad instance.

Maybe as Monad

```
instance Monad Maybe where
    -- return :: a -> Maybe a
    return x  =  Just x

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

This way, one can write alternations in connection with `Maybe` types more simply.

The `Maybe` monad "carries" a value that is accessible explicitly via `(>>=)`.

Maybe as Monad (2)

Extracted from the code of a reduction machine: here, `red_Redex_LMHead` reduces the condition in an alternation or reports failure with `Nothing`.

The alternation:

```
r (If c t e) = ... case red_Redex_LMHead c of
                      Just c' -> Just (If c' t e)
                      Nothing -> Nothing
```

can also be written:

```
r (If c t e) = ... do c' <- red_Redex_LMHead c
                       return (If c' t e)
```

or, with `>>=` rather than `do`:

```
r (If c t e) = ... red_Redex_LMHead c >>=
                        (\c' -> return (If c' t e))
```


[·] as Monad

Lists also form a monad:

```
instance Monad [] where
   -- return :: a -> [a]
   return x = [x]

   -- (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f  =  concat (map f xs)
```

This way, one can define, e.g., the list of all `True`/`False` combinations of length `n` as follows:

```
combs :: Int -> [[Bool]]
combs 0 = return []    -- or: [[]]
combs n = do { xs <- combs (n-1); [False:xs, True:xs] }
```

The Input/Output Problem in Functional Programming

- In SML:

  `output(std_out, "ha"); output(std_out, "ha")` $\longrightarrow$ `haha`

  `let val x = output(std_out, "ha") in x; x end` $\longrightarrow$ `ha`

  Referential transparency is compromised!

- Additional problem in Haskell:

  Laziness deprives the programmer of the control over the point in time at which the input/output takes place.

- Solution with monads: composition of monadic operations solely with operator `(>>=)` $\rightsquigarrow$ implementation of `(>>=)` can enforce deterministic computing.

Motivation for Monads for I/O and States

- Nailing down the order of computations
  - reasonable input/output operations in the presence of laziness
  - re-assignment for special, monadic arrays (numeric computation)
  - explicit deletion of data objects no longer needed
- Use of states
  - comfortable handling of global data objects
  - separation of cross-cutting concerns, e.g., logging, debugging, exception handling
  - generation of fresh names
  - graph algorithms: marking, identification of structures

Difference to Imperative Programming

- Embedding of non-monadic operations, recognizable by type
- Monadic computations can be composed functionally
  - monad combinators, e.g., `mapM`, `foldM`
  - monadic operations are themselves manipulable program values
  - distinction of these values and run-time values
- Possible dependences can be limited:
  a fixed monad can only implement certain operations
- Local use of monads (not the `IO` monad)
  initialization, use, termination
- Monads can be nested
- A monad can provide a specific service or feature

IO Monad

- Type constructor `IO`
- `return :: a -> IO a`
- `(>>=)  :: IO a -> (a -> IO b) -> IO b`
- Interpretation of type `IO a`: a *computation* with input/output and return value of type `a`
- Return value of a computation only to be used in a further computation, not as a general function result (for exceptional cases, there is an "unsafe" predefined function that overrides this requirement).
- Intuition: operations in the `IO` monad alter the outside world (side-effect).

Input/Output and Side-Effects

- What does input/output with side-effects mean?
  1. Loss of referential transparency?

     `getChar` does not always return the same value:

     ```
     Prelude> do { a<-getChar; b<-getChar; print (a==b) }
     12False
     ```
  2. Problem in a language with lazy evaluation:

     order of evaluation determines order of I/O operations
- Solution in Haskell:
  1. value of `getChar` is not of type `Char` but of type `IO Char`, i.e., an I/O operation
  2. order enforced by sequencing in the `IO` monad `(>>=)`

Referential Transparency with Monads

Referential transparency is preserved

- Formal view: the value of a monadic expression is not the return value as such
- Programmer's view:
  - strong resemblance with sequencing in imperative languages:
    compositional view can be lost and replaced subconsciously
    by operational thinking
    warning sign: long sequences in the program text, tail recursion
  - same risks as with imperative programming:
    aliasing, redundancy, inconsistency
  - avoidance of the risks by:
    monadic combinators, access functions for states

Predefined Input/Output Operations

- Input of a character from stdin: `getChar :: IO Char`
- Input of a file: `readFile :: FilePath -> IO String`
- Output on stdout: `putStr :: String -> IO ()`

  `putStrLn :: String -> IO ()`

  `print :: Show a => a -> IO ()`
- Output of a file: `writeFile :: FilePath -> String -> IO ()`
- and many more (see documentation)

The main function (entry point) of a compiled Haskell program is by default `Main.main` and must be of type `IO ()`. With GHC's option `-main-is X.f` one can make `X.f` the main function.

Monad Combinators

```
copyFile :: (String,String) -> IO ()
copyFile (x,y) = do content <- readFile x
                    writeFile y content
main :: IO ()
main = do copyFile ("input1","output1")
          copyFile ("input2","output2")
```

Aggregation via `mapM`:

```
main :: IO ()
main = do mapM copyFile [ ("input"++show i,"output"++show i)
                        | i<-[1..2] ]
          return ()
```

Central Monad Combinators (Module `Monad`)

```
when  :: Monad m => Bool -> m () -> m ()

mapM  :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()

sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()

foldM  :: Monad m => (a -> b -> m a) -> a -> [b] -> m a

liftM  :: Monad m => (a -> b) -> m a -> m b
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
ap     :: Monad m => m (a -> b) -> m a -> m b

join   :: Monad m => m (m a) -> m a
```


mapM and foldM (1)

```
import Monad
foo :: Int -> IO Int
foo i = do putStr ("foo " ++ show i ++ " called\n")
           return (i*i)
```

```
bar :: Int -> Int -> IO Int
bar i j = do putStr ("bar " ++ show i ++ " " ++ show j ++ " called\n")
             return (i+j)
```

```
main :: IO ()
main = do
        xs <- mapM foo [1..4]
        putStr ("point A: "++show xs++"\n")
        y <- foldM bar 0 [1..4]
        putStr ("point B: "++show y++"\n")
        return ()
```

`mapM` and `foldM` (2)

- Both `mapM` and `foldM` apply the functions applied to the elements of the list in sequence.
- `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`
  - argument of the ith operation is the ith element of list `[1..4]`
  - result of the ith operation becomes the ith element of list `xs`
- `foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a`
  - arguments of the ith operation are the return value of the $(i-1)$st operation (or the initial element 0) and the ith element of list `[1..4]`
  - `y` is the return value of the final operation

mapM and foldM (3)

Output of program foo/bar:

```
Main> main
foo 1 called                -- xs <- mapM foo [1..4]
foo 2 called
foo 3 called
foo 4 called
point A: [1,4,9,16]         -- putStr ("point A: "++show xs++"\n")
-----------------------------------------------------------------------
bar 0 1 called              -- y <- foldM bar 0 [1..4]
bar 1 2 called
bar 3 3 called
bar 6 4 called
point B: 10                 -- putStr ("point B: "++show y++"\n")
```

Complete Type Class Monad

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

    -- Minimal complete definition: (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  =  error s
```

Failure of the computation can be reported with `fail`.

State Monad

We would like to define a monad `State s` that stores values of type `s` (as background information). That is, a monadic computation of this monad is of type `State s a`: it returns a result of type `a` and possibly modifies the state of type `s`. Such a computation is, thus, equivalent to a (purely functional) computation of type `s -> (a,s)`.

```
newtype State s a = S (s -> (a,s))

instance Monad (State s) where
    -- return :: a -> State s a
    return x = S (\st -> (x,st))
    -- (>>=) :: State s a -> (a -> State s b) -> State s b
    S f >>= g  =  S (\st -> let (x,st') = f st
                                S h = g x
                            in h st'
                    )
```

Operations on the State Monad

```
-- return the state as the result of the computation
get :: State s s
get = S (\st -> (st, st))

-- replace the state by st
put :: s -> State s ()
put st = S (\_ -> ((), st))

-- compute (S f) in initial state inState
run :: s -> State s a -> a
run inState (S f) = let (result,_) = f inState in result
```

Exemplary Use of the State Monad

```
-- replace str by a new name if str == torepl
replace :: String -> String -> State [String] String
replace torepl str
    | str == torepl  =  do names <- get
                           let fresh = head names
                           put (tail names)
                           return fresh
    | otherwise      =  return str

-- replace all instances of torepl in strings by new names
replaceAll :: String -> [String] -> [String]
replaceAll torepl strings =
    run inState (mapM (replace torepl) strings)
    where
      inState = ['_' : show i | i <- [0..]]
```

Parsing with Monads

A parsing monad is a state monad whose state is the input not yet processed. The operations of the parsing monad "consume" the input in steps.

The *Parsec* library (module `Text.ParserCombinators.Parsec`) offers combinators for monadic parsing. The following slides summarize the basic functions. We omit, e.g., token recognition (scanning) or the monad transformer `ParsecT`.

We use only parsing operations of type `Parser a` that parse a `String` and return a result of type `a`. `Parser a` is actually a type synonym that is based on a type for more general parsers.

Some Combinators in Parsec

`char :: Char -> Parser Char`	recognizes exactly one character
`string :: String -> Parser String`	recognizes a character string
`letter, digit :: Parser Char`	recognizes a letter or digit
`eof :: Parser ()`	end of input
`many  :: Parser a -> Parser [a]`	apply a parser repeatedly, at least once
`many1 :: Parser a -> Parser [a]`	repeat a parser at least once
`between :: Parser open -> Parser close -> Parser a -> Parser a`	put a parser between two parsers
`(<|>) :: Parser a -> Parser a -> Parser a`	alternative

Initiate the parser with

```
parse :: Parser a -> SourceName -> String -> Either ParseError a
```

`SourceName` is a file name for error messages.

Recognition of the Languages $\{a^n b^n \mid n \geq 0\}$

`Parsec` offers several ways of recognizing the languages $\{a^n b^n \mid n \geq 0\}$. Since it is embedded in Haskell, `Parsec` can recognize more than context-free languages. With the parsers `aNbN4` and `aNbN5`, one can also recognize, e.g., the (context-sensitive) languages $\{a^n b^n c^n \mid n \geq 0\}$.

```
aNbN1 :: Parser ()
aNbN1 = do { char 'a'; aNbN1; char 'b'; return () }  <|>  return ()

aNbN2 :: Parser ()
aNbN2 = between (char 'a') (char 'b') aNbN2  <|>  return ()

aNbN3 :: Parser Int
aNbN3 = do { char 'a'; l <- aNbN3; char 'b'; return (l+1) }
        <|>  return 0
```

Recognition of the Languages $\{a^n b^n \mid n \geq 0\}$ (2)

```
aNbN4 :: Parser Int
aNbN4 = do
  as <- many (char 'a')
  let l = length as
  bs <- many (char 'b')
  when (length bs /= l) $ fail "number of a's and b's does not match"
  return l

aNbN5 :: Parser Int
aNbN5 = do
  as <- many (char 'a')
  let l = length as
  sequence_ [char 'b' | _ <- [1..l]]
  return l
```

Parsec: Benefits and Drawbacks

Benefits:

- enables parsing of context-sensitive languages
- is not a separate tool (parser generator)

Drawbacks:

- cannot handle left recursion in the grammar:

  `p = do { p; ... }` does not terminate

  $\rightsquigarrow$ eliminate left recursion!

  But: there are combinators for common cases, e.g.,

  `sepBy :: Parser a -> Parser sep -> Parser [a]`

  for enumeration with some separator

- Left-factorization required:

  for alternatives with shared prefix, backtracking with

  `try :: Parser a -> Parser a`

Combination of Monads

The combination of two or more monads is facilitated by doing the following for every monad:

- isolate the specific operations in a dedicated type class,
- define a *monad transformer* that specifies the combination of the monad with another monad.

Example from the *Monad Transformer Library* (MTL), operations for state manipulation:

```
class Monad m => MonadState s m where
   get :: m s
   put :: s -> m a

-- monad transformer StateT
newtype StateT s m a
evalStateT :: Monad m => StatetT s m a -> s -> m a
```

MTL: `ExceptT` and `Identity`

Operations for error handling:

```
class Monad m => MonadError e m where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
-- monad transformer ExceptT
newtype ExceptT e m a
runExceptT :: ExceptT e m a -> m (Either e a)
```

The basis for this use of the transformer is the identity monad:

```
-- identity monad (that does nothing)
newtype Identity a
runIdentity :: Identity a -> a
```

Example: Combination of StateT and ExceptT

```
data Exp = Const Int                    -- Constant
         | Var String                   -- Variable
         | Add Exp Exp                  -- Sum
         | Let (String,Exp) Exp         -- let x=e in a
           deriving Show

type Env = [(String,Int)]

eval :: (MonadState Env m, MonadError String m) => Exp -> m Int
eval (Const i) = return i
eval (Var x) = do
    env <- get
    case lookup x env of
      Just val -> return val
      Nothing  -> throwError ("variable '" ++ x ++ "' undefined")
```

```
eval (Add a b) = do
    c <- eval a
    d <- eval b
    return (c+d)
eval (Let (x,e) a) = do
    val <- eval e
    env <- get
    put ((x,val):env)            -- add (x,val) to environment
    res <- eval a
    put env                      -- restore old environment
    return res

doEval :: Env -> Exp -> Either String Int
doEval env exp = runIdentity $ runExceptT $ evalStateT (eval exp) env
```


Remarks on Transformers

- In general, the nesting of monads is not commutative.
- To enable arbitrary nestings, more instance declarations than shown here are required.
- In a monad nest, the `IO` monad (if used) must be innermost.