MGS 2006: AFP Lectures 1 & 2

Introduction to Monads

Henrik Nilsson

University of Nottingham, UK

Monads (1)

"Shall I be pure or impure?" (Wadler, 1992)

- Absence of effects
  - makes programs easier to understand and reason about
  - make lazy evaluation viable
  - enhances modularity and reuse.
- Effects (state, exceptions, . . .) can
  - yield concise programs
  - facilitate modifications
  - improve the efficiency.

Monads (2)

- Monads bridges the gap: allow effectful programming in a pure setting.
- ***Thus we shall be both pure and impure, whatever takes our fancy!***
- Monads originated in Category Theory.
- Adapted by Moggi for structuring denotational semantics.
- Adapted by Wadler for structuring functional programs.

Monads (3)

- Key idea of monads: computations as ***first-class entities***.
- Monads promotes disciplined, modular use of effects since the type of a program reflects which effects that occurs.
- Monads allows us great flexibility in tailoring the effect structure to our precise needs.

First Two Lectures

- Effectful computations: motivating examples
- Monads
- The Haskell do-notation
- Some standard monads
- A concurrency monad

Example: A Simple Evaluator

```
data Exp = Lit Integer
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp

eval :: Exp -> Integer
eval (Lit n)     = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 ‘div‘ eval e2
```


Making the evaluator safe (1)

```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 + n2)
```


Making the evaluator safe (2)

```
safeEval (Sub e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 - n2)
```

Making the evaluator safe (3)

```
safeEval (Mul e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 * n2)
```


Making the evaluator safe (4)

```
safeEval (Div e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 ->
                    if n2 == 0
                    then Nothing
                    else Just (n1 ‘div‘ n2)
```


Any common pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- Sequencing of evaluations.
- If one evaluation fail, fail overall.
- Otherwise, make result available to following evaluations.

Sequencing evaluations (1)

```
evalSeq :: Maybe Integer
           -> (Integer -> Maybe Integer)
           -> Maybe Integer
evalSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

Sequencing evaluations (2)

```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    safeEval e1 ‘evalSeq‘ (\n1 ->
    safeEval e2 ‘evalSeq‘ (\n2 ->
    Just (n1 + n2)))
safeEval (Sub e1 e2) =
    safeEval e1 ‘evalSeq‘ (\n1 ->
    safeEval e2 ‘evalSeq‘ (\n2 ->
    Just (n1 - n2)))
```


Sequencing evaluations (3)

```
safeEval (Mul e1 e2) =
    safeEval e1 ‘evalSeq‘ (\n1 ->
    safeEval e2 ‘evalSeq‘ (\n2 ->
    Just (n1 - n2)))
safeEval (Div e1 e2) =
    safeEval e1 ‘evalSeq‘ (\n1 ->
    safeEval e2 ‘evalSeq‘ (\n2 ->
    if n2 == 0
    then Nothing
    else Just (n1 ‘div‘ n2)))
```


Aside: Scope rules of λ-abstractions

The scope rules of λ-abstractions are such that parentheses can be omitted:

```
safeEval :: Exp -> Maybe Integer
...
safeEval (Add e1 e2) =
    safeEval e1 ‘evalSeq‘ \n1 ->
    safeEval e2 ‘evalSeq‘ \n2 ->
    Just (n1 + n2)
...
```


Exercise 1: Inline `evalSeq` (1)

```
safeEval (Add e1 e2) =
  safeEval e1 ‘evalSeq‘ \n1 ->
  safeEval e2 ‘evalSeq‘ \n2 ->
  Just (n1 + n2)
```

$=$

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just a -> (\n1 -> safeEval e2 ...) a
```

Exercise 1: Inline `evalSeq` (2)

=

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> safeEval e2 ‘evalSeq‘ (\n2 -> ...)
```

=

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
                 Nothing -> Nothing
                 Just a -> (\n2 -> ...) a
```


Exercise 1: Inline `evalSeq` (3)

=

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
                 Nothing -> Nothing
                 Just n2 -> (Just n1 + n2)
```


Maybe viewed as a computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type `a` that ***may fail***.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
- I.e. ***failure is an effect***, implicitly affecting subsequent computations.
- Let’s generalize and adopt names reflecting our intentions.

Maybe viewed as a computation (2)

Successful computation of a value:

```
mbReturn :: a -> Maybe a
mbReturn = Just
```

Sequencing of possibly failing computations:

```
mbSeq :: Maybe a -> (a -> Maybe b) -> Maybe b
mbSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

Maybe viewed as a computation (3)

Failing computation:

```
mbFail :: Maybe a
mbFail = Nothing
```


The safe evaluator revisited

```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = mbReturn n
safeEval (Add e1 e2) =
    safeEval e1 ‘mbSeq‘ \n1 ->
    safeEval e2 ‘mbSeq‘ \n2 ->
    mbReturn (n1 + n2)
...
safeEval (Div e1 e2) =
    safeEval e1 ‘mbSeq‘ \n1 ->
    safeEval e2 ‘mbSeq‘ \n2 ->
    if n2 == 0 then mbFail
    else mbReturn (n1 ‘div‘ n2)))
```


Example: Numbering trees

```
data Tree a = Leaf a | Tree a :ˆ: Tree a

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
    where
        ntAux (Leaf _)     n = (Leaf n, n+1)
        ntAux (t1 :ˆ: t2) n =
            let (t1’, n’)  = ntAux t1 n
            in let (t2’, n’’) = ntAux t2 n’
               in (t1’ :ˆ: t2’, n’’)
```


Observations

- Repetitive pattern: threading a counter through a ***sequence*** of tree numbering ***computations***.
- It is very easy to pass on the wrong version of the counter!

Can we do better?

Stateful Computations (1)

- A ***stateful computation*** consumes a state and returns a result along with a possibly updated state.
- The following type synonym captures this idea:

```
type S a = Int -> (a, Int)
```

  (Only `Int` state for the sake of simplicity.)
- A value (function) of type `S a` can now be viewed as denoting a stateful computation computing a value of type `a`.

Stateful Computations (2)

- When sequencing stateful computations, the resulting state should be passed on to the next computation.
- I.e. ***state updating is an effect***, implicitly affecting subsequent computations.
  (As we would expect.)

Stateful Computations (3)

Computation of a value without changing the state:

```
sReturn :: a -> S a
sReturn a = \n -> (a, n)
```

Sequencing of stateful computations:

```
sSeq :: S a -> (a -> S b) -> S b
sSeq sa f = \n ->
    let (a, n') = sa n
    in f a n'
```


Stateful Computations (4)

Reading and incrementing the state:

```
sInc :: S Int
sInc = \n -> (n, n + 1)
```

Numbering trees revisited

```
data Tree a = Leaf a | Tree a :^: Tree a

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
    where
        ntAux (Leaf _)  =
            sInc `sSeq` \n -> sReturn (Leaf n)
        ntAux (t1 :^: t2) =
            ntAux t1 `sSeq` \t1' ->
            ntAux t2 `sSeq` \t2' ->
            sReturn (t1' :^: t2')
```


Observations

- The “plumbing” has been captured by the abstractions.
- In particular, there is no longer any risk of “passing on” the wrong version of the state!

Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing identically structured abstractions that encapsulated the effects:
  - A type denoting computations
  - A combinator for computing a value without any effect
  - A combinator for sequencing computations
- In fact, both examples are instances of the general notion of a ***MONAD***.

Monads in Functional Programming

A monad is represented by:

- A type constructor

  `M :: * -> *`

  `M T` represents computations of a value of type `T`.
- A polymorphic function

  `return :: a -> M a`

  for lifting a value to a computation.
- A polymorphic function

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

  for sequencing computations.

Exercise 2: join and fmap

Equivalently, the notion of a monad can be captured through the following functions:

```
return :: a -> M a
join :: (M (M a)) -> M a
fmap :: (a -> b) -> (M a -> M b)
```

`join` "flattens" a computation, `fmap` "lifts" a function to map computations to computations.

Define `join` and `fmap` in terms of >>= (and `return`), and >>= in terms of `join` and `fmap`.

Exercise 2: Solution

```
join :: M (M a) -> M a
join mm = mm >>= id

fmap :: (a -> b) -> M a -> M b
fmap f m = m >>= \x -> return (f x)

(>>=) :: M a -> (a -> M b) -> M b
m >>= f = join (fmap f m)
```


Monad laws

Additionally, some simple laws must be satisfied:

$$\begin{aligned} \texttt{return}\ x \mathbin{\texttt{>>=}} f &= f\ x \\ m \mathbin{\texttt{>>=}} \texttt{return} &= m \\ (m \mathbin{\texttt{>>=}} f) \mathbin{\texttt{>>=}} g &= m \mathbin{\texttt{>>=}} (\lambda x \rightarrow f\ x \mathbin{\texttt{>>=}} g) \end{aligned}$$

I.e., `return` is the right and left identity for >>=, and >>= is associative.

Exercise 3: The Identity Monad

The ***Identity Monad*** can be understood as representing ***effect-free*** computations:

```
type I a = a
```

1. Provide suitable definitions of `return` and >>=.
2. Verify that the monad laws hold for your definitions.

Exercise 3: Solution

```
return :: a -> I a
return = id

(>>=) :: I a -> (a -> I b) -> I b
m >>= f = f m
-- or: (>>=) = flip ($)
```

Simple calculations verify the laws, e.g.:

$$\begin{aligned} \texttt{return}\ x \ \texttt{>>=}\ f &= \texttt{id}\ x \ \texttt{>>=}\ f \\ &= x \ \texttt{>>=}\ f \\ &= f\ x \end{aligned}$$

Monads in Category Theory (1)

The notion of a monad originated in Category Theory. There are several equivalent definitions (Benton, Hughes, Moggi 2000):

- ***Kleisli triple/triple in extension form:*** Most closely related to the >>= version:

  A ***Klesili triple*** over a category $\mathcal{C}$ is a triple $(T, \eta, \_^*)$, where $T : |\mathcal{C}| \to |\mathcal{C}|$, $\eta_A : A \to TA$ for $A \in |\mathcal{C}|$, $f^* : TA \to TB$ for $f : A \to TB$.

  (Additionally, some laws must be satisfied.)

Monads in Category Theory (2)

- ***Monad/triple in monoid form:*** More akin to the `join`/`fmap` version:

  A ***monad*** over a category $\mathcal{C}$ is a triple (T, η, μ), where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\eta : \mathrm{id}_{\mathcal{C}} \dot{\to} T$ and $\mu : T^2 \dot{\to} T$ are natural transformations.

  (Additionally, some commuting diagrams must be satisfied.)

Monads in Haskell (1)

In Haskell, the notion of a monad is captured by a ***Type Class***:

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

This allows the names of the common functions to be overloaded, and the sharing of derived definitions.

Monads in Haskell (2)

The Haskell monad class have two further methods with default instances:

```
(>>) :: m a -> m b -> m b
m >> k = m >>= \_ -> k

fail :: String -> m a
fail s = error s
```


The Maybe monad in Haskell

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

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


Exercise 4: A state monad in Haskell

Haskell 98 does not permit type synonyms to be instances of classes. Hence we have to define a new type:

```
newtype S a = S (Int -> (a, Int))

unS :: S a -> (Int -> (a, Int))
unS (S f) = f
```

Provide a `Monad` instance for `S`.

Exercise 4: Solution

```
instance Monad S where
    return a = S (\s -> (a, s))

    m >>= f = S $ \s ->
        let (a, s') = unS m s
        in unS (f a) s'
```

Monad-specific operations (1)

To be useful, monads need to be equipped with additional operations specific to the effects in question. For example:

```
fail :: String -> Maybe a
fail s = Nothing

catch :: Maybe a -> Maybe a -> Maybe a
m1 ‘catch‘ m2 =
    case m1 of
        Just _  -> m1
        Nothing -> m2
```


Monad-specific operations (2)

Typical operations on a state monad:

```
set :: Int -> S ()
set a = S (\_ -> ((), a))

get :: S Int
get = S (\s -> (s, s))
```

Moreover, there is often a need to “run” a computation. E.g.:

```
runS :: S a -> a
runS m = fst (unS m 0)
```


The do-notation (1)

Haskell provides convenient syntax for programming with monads:

do
    a <- exp_1
    b <- exp_2
    return exp_3

is syntactic sugar for

exp_1 >>=\a ->
exp_2 >>=\b ->
return exp_3

The do-notation (2)

Computations can be done solely for effect, ignoring the computed value:

do
    exp_1
    exp_2
    return exp_3

is syntactic sugar for

exp_1 >>=_ ->
exp_2 >>=_ ->
return exp_3

The do-notation (3)

A `let`-construct is also provided:

```
do
    let a = exp1
        b = exp2
    return exp3
```

is equivalent to

```
do
    a <- return exp1
    b <- return exp2
    return exp3
```


Numbering trees in do-notation

```
numberTree :: Tree a -> Tree Int
numberTree t = runS (ntAux t)
    where
        ntAux (Leaf _) = do
            n <- get
            set (n + 1)
            return (Leaf n)
        ntAux (t1 :^: t2) = do
            t1' <- ntAux t1
            t2' <- ntAux t2
            return (t1' :^: t2')
```


Monadic utility functions

Some monad utilities, some from the Prelude, some from the module Monad:

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
mapM      :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_     :: Monad m => (a -> m b) -> [a] -> m ()
when      :: Monad m => Bool -> m () -> m ()
foldM     :: Monad m =>
                 (a -> b -> m a) -> a -> [b] -> m a
liftM     :: Monad m => (a -> b) -> (m a -> m b)
```


Exercise 5: Monadic utilities

Define

```
when     :: Monad m => Bool -> m () -> m ()
sequence :: Monad m => [m a] -> m [a]
mapM     :: Monad m => (a -> m b) -> [a] -> m [b]
```

in terms of the basic monad functions.

Exercise 5: Solution (1)

```
when :: Monad m => Bool -> m () -> m ()
when p m = if p then m else return ()

sequence  :: Monad m => [m a] -> m [a]
sequence []        = return []
sequence (ma:mas) = ma >>= \a ->
                    sequence mas >>= \as ->
                    return (a:as)
```


Exercise 5: Solution (2)

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (a:as) = f a >>= \b ->
                mapM f as >>= \bs ->
                return (b:bs)
```


The Haskell IO monad

In Haskell, IO is handled through the IO monad. IO is ***abstract***! Conceptually:

```
newtype IO a = IO (World -> (a, World))
```

Some operations:

```
putChar     :: Char -> IO ()
putStr      :: String -> IO ()
putStrLn    :: String -> IO ()
getChar     :: IO Char
getLine     :: IO String
getContents :: String
```


The ST Monad: “real” state

The ST monad (common Haskell extension) provides real, imperative state behind the scenes to allow efficient implementation of imperative algorithms:

```
data ST s a -- abstract
instance Monad (ST s)

newSTRef   :: s ST a (STRef s a)
readSTRef  :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()

runST :: (forall s . st s a) -> a
```

Nondeterminism: The list monad

```
instance Monad [] where
    return a = [a]
    m >>= f  = concat (map f m)
    fail s   = []
```

Example:

```
do
    x <- [1, 2]
    y <- ['a', 'b']
    return (x,y)
```

Result: `[(1,'a'),(1,'b'),(2,'a'),(2,'b')]`

Environments: The reader monad

```
instance Monad ((->) e) where
    return a = const a
    m >>= f  = \e -> f (m e) e

getEnv :: ((->) e) e
getEnv = id
```

Cf. the combinators S, K, and I!

```
I :: a -> a
K :: a -> b -> a
S :: (a -> b -> c) -> (a -> b) -> a -> c
(>>=) :: (a -> b) -> (b -> a -> c) -> a -> c
```


The continuation monad (1)

- In Continuation-Passing style (CPS), a ***continuation*** representing the “rest of the computation” is passed to each computation.
- A continuation is a function that when applied to the result of the current subcomputation, returns the final result of the overall computation.
- Making continuations explicitly available makes it possible to implement control-flow effects, like jumps.

The continuation monad (2)

```
data CPS r a = CPS ((a -> r) -> r)

unCPS :: CPS r a -> ((a -> r) -> r)
unCPS (CPS f) = f

instance Monad (CPS r) where
    return a = CPS (\k -> k a)
    m >>= f  = CPS $ \k ->
        unCPS m (\a -> unCPS (f a) k)
```

The continuation monad (3)

```
callCC :: ((a -> CPS r b) -> CPS r a) -> CPS r a
callCC f = CPS $ \k ->
    unCPS (f (\a -> CPS (\_ -> k a))) k

runCPS :: CPS a a -> a
runCPS m = unCPS m id
```


Exercise 6: Control transfer

```
f :: Int -> Int -> Int
f x y = runCPS $ do
    callCC $ \exit -> do
        let d = x - y
        when (d == 0) (exit (-1))
        let z = (abs ((x + y) `div` d))
        when (z > 10) (exit (-2))
        return (z^3)
```

Compute `f 10 6`, `f 10 10`, and `f 10 9`.

A Concurrency Monad (1)

A `Thread` represents a process: a stream of primitive ***atomic*** operations:

```
data Thread = Print Char Thread
            | Fork Thread Thread
            | End
```

Note that a `Thread` represents the ***entire rest*** of a computation.

A Concurrency Monad (2)

Introduce a monad representing “interleavable computations”. At this stage, this amounts to little more than a convenient way to construct threads by sequential composition.

How can `Thread`s be composed sequentially? The only way is to parameterize thread prefixes on the rest of the `Thread`. This leads directly to ***continuations***.

A Concurrency Monad (3)

```
newtype CM a = CM ((a -> Thread) -> Thread)

fromCM :: CM a -> ((a -> Thread) -> Thread)
fromCM (CM x) = x

thread :: CM a -> Thread
thread m = fromCM m (const End)

instance Monad CM where
    return x = CM (\k -> k x)
    m >>= f  = CM $ \k ->
         fromCM m (\x -> fromCM (f x) k)
```


A Concurrency Monad (4)

Atomic operations:

```
cPrint :: Char -> CM ()
cPrint c = CM (\k -> Print c (k ()))

cFork :: CM a -> CM ()
cFork m = CM (\k -> Fork (thread m) (k ()))

cEnd :: CM a
cEnd = CM (\_ -> End)
```


A Concurrency Monad (5)

Running a computation:

```
type Output = [Char]
type ThreadQueue = [Thread]
type State = (Output, ThreadQueue)

runCM :: CM a -> Output
runCM m = runHlp ("", []) (thread m)
    where
        runHlp s t =
            case dispatch s t of
                Left (s', t) -> runHlp s' t
                Right o      -> o
```


A Concurrency Monad (6)

Dispatch on the operation of the currently running `Thread`. Then call the scheduler.

```
dispatch :: State -> Thread
            -> Either (State, Thread) Output
dispatch (o, rq) (Print c t) =
    schedule (o ++ [c], rq ++ [t])
dispatch (o, rq) (Fork t1 t2) =
    schedule (o, rq ++ [t1, t2])
dispatch (o, rq) End =
    schedule (o, rq)
```

A Concurrency Monad (7)

Selects next `Thread` to run, if any.

```
schedule :: State -> Either (State, Thread)
                            Output
schedule (o, [])    = Right o
schedule (o, t:ts) = Left ((o, ts), t)
```


Example: Concurrent processes

```
p1 :: CM ()          p2 :: CM ()        p3 :: CM ()
p1 = do              p2 = do            p3 = do
    cPrint 'a'           cPrint '1'       cFork p1
    cPrint 'b'           cPrint '2'       cPrint 'A'
    ...                  ...              cFork p2
    cPrint 'j'           cPrint '0'       cPrint 'B'

main = print (runCM p3)
```

Result: `aAbc1Bd2e3f4g5h6i7j890`
(As it stands, the output is only made available after ***all*** threads have terminated.)

Alternative version

Incremental output:

```
runCM :: CM a -> Output
runCM m = dispatch [] (thread m)

dispatch :: ThreadQueue -> Thread -> Output
dispatch rq (Print c t)  = c : schedule (rq ++ [t]
dispatch rq (Fork t1 t2) = schedule (rq ++ [t1, t2
dispatch rq End          = schedule rq

schedule :: ThreadQueue -> Output
schedule []     = []
schedule (t:ts) = dispatch ts t
```


Example: Concurrent processes 2

```
p1 :: CM ()          p2 :: CM ()        p3 :: CM ()
p1 = do              p2 = do            p3 = do
    cPrint 'a'           cPrint '1'       cFork p1
    cPrint 'b'           undefined        cPrint 'A'
    ...                  ...              cFork p2
    cPrint 'j'           cPrint '0'       cPrint 'B'

main = print (runCM p3)
```

Result: aAbc1Bd`*** Exception: Prelude.undefined`

Reading

- Nomaware. *All About Monads.* `http://www.nomaware.com/monads`
- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- Koen Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9(3), 1999.
- Philip Wadler. The Essence of Functional Programming. *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, 1992.