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 \( \text{\texttt{\textasciicircum}} \) 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)

```haskell
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)

```haskell
evalSeq :: Maybe Integer 
  -> (Integer -> Maybe Integer) 
  -> Maybe Integer

evalSeq ma f = 
  case ma of 
    Nothing -> Nothing 
    Just a -> f a
```

Sequencing evaluations (2)

```haskell
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)

```haskell
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 \-abstractions

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

```haskell
safeEval :: Exp -> Maybe Integer 

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

Exercise 1: Inline evalSeq (1)

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

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

Exercise 1: Inline evalSeq (2)

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

Exercise 1: Inline evalSeq (3)

```haskell
= 
  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 \( \text{Maybe} \ a \) 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.

The safe evaluator revisited

```haskell
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = mbReturn n
safeEval (Add e1 e2) =
  safeEval e1 'mbSeq' \n1 ->
  safeEval e2 'mbSeq' \n2 ->
  mbReturn (n1 + n2)
...
```

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:
  ```haskell
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 \( a \).

Successful computation of a value:

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

Sequencing of possibly failing computations:

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

Example: Numbering trees

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

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
where
  ntAux (Leaf _) n = (Leaf n, n+1)
  ntAux (t1 :^: t2) n =
    let (t1', n') = ntAux t1 n
    in let (t2', n'') = ntAux t2 n'
    in (t1' :^: 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 (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:
  ```haskell
  sReturn :: a -> S a
  sReturn a = \n -> (a, n)
```
- Sequencing of stateful computations:
  ```haskell
  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:

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

Numbering trees revisited

```haskell
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
  ```haskell
  M :: * -> *
  M T
  represents computations of a value of type T.
  ```
- A polymorphic function
  ```haskell
  return :: a -> M a
  ```
  for lifting a value to a computation.
- A polymorphic function
  ```haskell
  (>>=) :: M a -> (a -> M b) -> M b
  ```
  for sequencing computations.

Additionally, some simple laws must be satisfied:

- **return** is the right and left identity for **(>>=)**,
- **(>>=)** is associative.

Exercise 2: Solution

```haskell
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)
```

Exercise 3: The Identity Monad

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

```haskell
type I a = a
1. Provide suitable definitions of return and **(>>=)**.
2. Verify that the monad laws hold for your definitions.
```
**Exercise 3: Solution**

\[
\begin{align*}
\text{return} :: & a \to I a \\
\text{return} = & \text{id} \\
(\gg\gg) :: & I a \to (a \to I b) \to I b \\
& \text{or: } (\gg\gg) = \text{flip } ($) \\
\end{align*}
\]

Simple calculations verify the laws, e.g.:\[
\begin{align*}
\text{return } x \gg\gg f & = \text{id } x \gg\gg f \\
& = x \gg\gg f \\
& = f x
\end{align*}
\]

**Monads in Category Theory (1)**

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

- **Klesili triple/triple in extension form:** Most closely related to the \(\gg\gg\) version:

  A *Klesili triple* over a category \(\mathcal{C}\) is a triple \((T, \eta, \mu)\), where \(T : \mathcal{C} \to \mathcal{C}\), \(\eta : A \to TA\) 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 \textit{join/fmap} version:

  A *monad* over a category \(\mathcal{C}\) is a triple \((T, \eta, \mu)\), where \(T : \mathcal{C} \to \mathcal{C}\) is a functor, \(\eta : \text{id}_\mathcal{C} \to T\) and \(\mu : T^2 \to T\) 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*:

\[
\begin{align*}
\text{class Monad } m \text{ where} \\
\text{return} :: & a \to m a \\
(\gg\gg) :: & m a \to (a \to m b) \to m b
\end{align*}
\]

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:

\[
\begin{align*}
(>>) :: & m a \to m b \to m b \\
\text{fail} :: & \text{String } \to m a \\
\text{fail } s & = \text{error } s
\end{align*}
\]

**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:

\[
\begin{align*}
\text{newtype } S a & = S \text{ Int } \to (a, \text{ Int}) \\
\text{unS} :: & S a \to (\text{Int } \to (a, \text{ Int})) \\
\text{unS} (S f) & = f
\end{align*}
\]

Provide a *Monad* instance for \(S\).

**Exercise 4: Solution**

\[
\begin{align*}
\text{instance Monad } S \text{ where} \\
\text{return} a & = S \text{ \{ \_ } \to (a, \_)) \\
\text{m } \gg\gg f & = S \text{ \{ \_ } \to \_ ) \\
\text{let } (a, s') & = \text{unS } m \text{ } s \\
\text{in } \text{unS } (f a) s'
\end{align*}
\]

**Monad-specific operations (1)**

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

\[
\begin{align*}
\text{fail} :: & \text{String } \to \text{Maybe } a \\
\text{fail } s & = \text{Nothing} \\
\text{catch} :: & \text{Maybe } a \to (\text{Maybe } a) \to \text{Maybe } a \\
\text{catch } \_ & = \text{Nothing} \\
\text{Just } x & \gg\gg f & = \text{f } x
\end{align*}
\]

**The Maybe monad in Haskell**

\[
\begin{align*}
\text{instance Monad } \text{Maybe} \text{ where} \\
\text{return} a & = \text{Just } a \\
\text{fail} & = \text{Nothing} \\
\text{fail } s & = \text{error } s
\end{align*}
\]

\[
\begin{align*}
\text{catch} :: & \text{Maybe } a \to (\text{Maybe } a) \to \text{Maybe } a \\
\text{catch } \_ & = \text{Nothing} \\
\text{Just } x & \gg\gg f & = f x
\end{align*}
\]

\[
\begin{align*}
\text{catch} :: & \text{Maybe } a \to (\text{Maybe } a) \to \text{Maybe } a \\
\text{catch } \_ & = \text{Nothing} \\
\text{Just } x & \gg\gg f & = f x
\end{align*}
\]
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
```

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')
```

Exercise 5: Monadic utility functions

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)
```

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: 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
```

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.

```
data CPS r a = CPS ((a -> CPS r b) -> CPS r a)
unCPS :: CPS r a -> ((a -> CPS r b) -> CPS r a)
instance Monad (CPS r) where
  return a = CPS \f -> f a
  m >>= f = CPS \k -> unCPS m \a -> unCPS (f a) k
runCPS :: CPS a a -> a
runCPS m = unCPS m id
```

Exercise 6: Control transfer

```
f :: Int -> Int -> Int
f x y = runCPS \f ->
  unCPS (f \(a -> CPS \(\_ -> k a)) k)
runCPS :: CPS a a -> a
runCPS m = unCPS m id
```

The continuation monad (3)

```
callCC :: ((a -> CPS r b) -> CPS r a) -> CPS r a
```

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.

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')]
```

The continuation monad (2)

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

```
unCPS :: CPS r a -> ((a -> CPS r b) -> CPS r a)
instance Monad (CPS r) where
  return a = CPS \f -> f a
  m >>= f = CPS \k ->
    unCPS m \a -> unCPS (f a) k
runCPS :: CPS a a -> a
runCPS m = unCPS m id
```

A Concurrency 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 \f -> f a
  m >>= f = CPS \k ->
    unCPS m \a -> unCPS (f a) k
runCPS :: CPS a a -> a
runCPS m = unCPS m id
```

Exercise 6: Control transfer

```
f :: Int -> Int -> Int
f x y = runCPS \f ->
  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 \f ->
  unCPS (f \(a -> CPS \(\_ -> k a)) k)
runCPS :: CPS a a -> a
runCPS m = unCPS m id
```

The continuation monad (3)

```
callCC :: ((a -> CPS r b) -> CPS r a) -> CPS r a
```

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.
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 threads be composed sequentially?
The only way is to parameterize thread prefixes on the rest of the thread. This leads directly to continuations.

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)

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

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)

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

Example: Concurrent processes 2

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

Main = print (runCM p3)
Result: aAbc18d2e3f4g5h6i7j890
(As it stands, the output is only made available after all threads have terminated.)

Example: Concurrent processes 2

Main = print (runCM p3)
Result: aAbc18d** Exception: Prelude.undefined
Reading

- Nomaware. All About Monads.  
  http://www.nomaware.com/monads


- Koen Claessen. A Poor Man’s Concurrency Monad.  