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.
- Key idea: Computational types: an object of type $MA$ denotes a computation of an object of type $A$.
- Thus we shall be both pure and impure, whatever takes our fancy!
- Monads originated in Category Theory.
- Adapted by
  - Moggi for structuring denotational semantics
  - Wadler for structuring functional programs

Monads (3)

- Monads
  - promote disciplined use of effects since the type reflects which effects can occur;
  - allow great flexibility in tailoring the effect structure to precise needs;
  - support changes to the effect structure with minimal impact on the overall program structure;
  - allow integration into a pure setting of “real” effects such as
    - I/O
    - mutable state.
First Two Lectures

- Effectful computations: motivating examples
- Monads
- The Haskell `do`-notation
- Some standard monads
- Monad transformers

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)

```
data Maybe a = Nothing | Just a

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 (or computations).
- If one evaluation fails, fail overall.
- Otherwise, make result available to following evaluations.

Example: Numbering trees

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

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

Sequencing evaluations (1)

**Sequencing** is common to both examples, with the outcome of a computation affecting subsequent computations.

```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)))
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)))
```

Sequencing evaluations (3)
Aside: Scope rules of λ-abstractions

The scope rules of λ-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)
...
```

Inlining `evalSeq` (1)

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

= 

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

Inlining `evalSeq` (2)

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

Excercise 1: Verify the other cases.
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 adopt names reflecting our intentions.

Maybe viewed as a computation (2)

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
```

Maybe viewed as a computation (3)

Failing computation:

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

The safe evaluator revisited

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

---

• 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.)

---

Computation of a value without changing the state:

```
sReturn :: a -> S a
sReturn a =
```

Sequencing of stateful computations:

```
sSeq :: S a -> (a -> S b) -> S b
sSeq sa f =
```

---

Reading and incrementing the state:

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

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

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
  where
    ntAux :: Tree a -> S (Tree Int)
    ntAux (Leaf _) =
      sInc sSeq \n -> sReturn (Leaf n)
    ntAux (Node t1 t2) =
      ntAux t1 sSeq \t1' ->
      ntAux t2 sSeq \t2' ->
      sReturn (Node 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:
  - A type denoting computations
  - A function constructing an effect-free computation of a value
  - A function constructing a computation by 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
  \[ M :: \ast \rightarrow \ast \]
  \[ M \ T \] represents computations of a value of type \( T \).
• A polymorphic function
  \[ \text{return} :: a \rightarrow M a \]
  for lifting a value to a computation.
• A polymorphic function
  \[ (\gg=) :: M a \rightarrow (a \rightarrow M b) \rightarrow M b \]
  for sequencing computations.
**Exercise 2: join and fmap**

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

\[
\begin{align*}
\text{return} & : \ a \rightarrow M \ a \\
\text{join} & : (M (M \ a)) \rightarrow M \ a \\
\text{fmap} & : (a \rightarrow b) \rightarrow (M \ a \rightarrow M \ b)
\end{align*}
\]

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.

**Monad laws**

Additionally, the following laws must be satisfied:

\[
\begin{align*}
\text{return} \ x \gg= f & = f \ x \\
\ m \gg= \text{return} & = m \\
\ (m \gg= f) \gg= g & = m \gg= (\lambda x \rightarrow f \ x \gg= g)
\end{align*}
\]

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

**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)

**Exercise 3: The Identity Monad**

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

\[
\text{type} \ I \ a = a
\]

1. Provide suitable definitions of return and >>=.
2. Verify that the monad laws hold for your definitions.
Exercise 3: Solution

\[
\text{return} :: \text{a} \rightarrow \text{I a} \\
\text{return} = \text{id}
\]

\[
(\gg\gg) :: \text{I a} \rightarrow (\text{a} \rightarrow \text{I b}) \rightarrow \text{I b} \\
\text{m} \gg\gg f = f \text{ m} \\
\text{-- or:} \ (\gg\gg) = \text{flip} \ (\_)
\]

Simple calculations verify the laws, e.g.:

\[
\text{return} \ x \gg\gg f = \text{id} \ x \gg\gg f = x \gg\gg f = f \ x
\]

Monads in Haskell (1)

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

\[
\text{class Monad m where} \\
\text{return} :: \text{a} \rightarrow m \text{ a} \\
(\gg\gg) :: m \text{ a} \rightarrow (\text{a} \rightarrow m \text{ b}) \rightarrow m \text{ b}
\]

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 has two further methods with default instances:

\[
(\gg) :: m \text{ a} \rightarrow m \text{ b} \rightarrow m \text{ b} \\
\text{m} \gg \text{k} = \text{m} \gg\gg \_ \rightarrow \text{k}
\]

\[
\text{fail} :: \text{String} \rightarrow m \text{ a} \\
\text{fail} \ s = \text{error} \ s
\]

The Maybe monad in Haskell

\[
\text{instance Monad Maybe where} \\
\text{-- return} :: \text{a} \rightarrow \text{Maybe a} \\
\text{return} = \text{Just} \\
\text{-- (\gg\gg)} :: \text{Maybe a} \rightarrow (\text{a} \rightarrow \text{Maybe b}) \rightarrow \text{Maybe b} \\
\text{Nothing} \gg\gg \_ = \text{Nothing} \\
(\text{Just} \ x) \gg\gg f = f \ x
\]
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:

```haskell
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

```haskell
instance Monad S where
  return a = S (
    s -> (a, s)
  )
  m >>= f = S $ \s ->
    let (a, s') = unS m
        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:

```haskell
fail :: String -> Maybe a
fail s = Nothing
catch :: Maybe a -> Maybe a -> Maybe a
ml `catch` m2 =
  case ml of
    Just _ -> m1
    Nothing -> m2
```

Monad-specific operations (2)

Typical operations on a state monad:

```haskell
set :: Int -> S ()
set a = S (
  _ -> ()
)
get :: S Int
get = S (
  s -> (s, s)
)```

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

```haskell
runS :: S a -> a
runS m = fst (unS m 0)
```
The do-notation (1)

Haskell provides convenient syntax for programming with monads:

```haskell
do 
  a <- exp_1 
  b <- exp_2 
  return exp_3
```

is syntactic sugar for

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

The do-notation (2)

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

```haskell
do 
  exp_1 
  exp_2 
  return exp_3
```

is syntactic sugar for

```haskell
exp_1 >>= \_ -> 
exp_2 >>= \_ -> 
return exp_3
```

The do-notation (3)

A let-construct is also provided:

```haskell
do 
  let a = exp_1 
  b = exp_2 
  return exp_3
```

is equivalent to

```haskell
do 
  a <- return exp_1 
  b <- return exp_2 
  return exp_3
```

Numbering trees in do-notation

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

Environments: The reader monad

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

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

Monad Transformers (1)

What if we need to support more than one type of effect?

For example: State and Error/Partiality?

We could implement a suitable monad from scratch:

newtype SE s a = SE (s -> Maybe (a, s))
Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been
  
  \[
  \text{newtype } \text{SE } s \ a = \text{SE } (s \to (\text{Maybe } a, s))
  \]

- Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.

Monad Transformers (3)

**Monad Transformers** can help:

- A **monad transformer** transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, ...), allowing the programmer to mix and match.
- A form of **aspect-oriented programming**.

Monad Transformers in Haskell (1)

- A monad transformer maps monads to monads. This is represented by a type constructor of the following kind:

  \[
  T :: (* \to *) \to (* \to *)
  \]

- Additionally, we require monad transformers to **add** computational effects. Thus we require a mapping from computations in the underlying monad to computations in the transformed monad:

  \[
  \text{lift} :: M \ a \to T \ M \ a
  \]

Monad Transformers in Haskell (2)

- These requirements are captured by the following (multi-parameter) type class:

  ```haskell
  class (Monad m, Monad (t m)) => MonadTransformer t m where
  lift :: m a -> t m a
  ```
Classes for Specific Effects

A monad transformer adds specific effects to any monad. Thus there can be many monads supporting the same operations. Introduce classes to handle the overloading:

```haskell
class Monad m => E m where
eFail :: m a
eHandle :: m a -> m a
```

```haskell
class Monad m => S m s | m -> s where
sSet :: s -> m ()
sGet :: m s
```

The Identity Monad

We are going to construct monads by successive transformations of the identity monad:

```haskell
newtype I a = I a
unI (I a) = a
instance Monad I where
return a = I a
m >>= f = f (unI m)
runI :: I a -> a
runI = unI
```

The Error Monad Transformer (1)

```haskell
newtype ET m a = ET (m (Maybe a))
unET (ET m) = m
instance Monad m => Monad (ET m) where
return a = ET (return (Just a))
m >>= f = ET $ do
  ma <- unET m
case ma of
    Nothing -> return Nothing
    Just a -> unET (f a)
runET :: Monad m => ET m a -> m a
runET etm = do
  ma <- unET etm
case ma of
    Just a -> return a
ET is a monad transformer:
instance Monad m => MonadTransformer ET m where
  lift m = ET (m >>= \a -> return (Just a))
```
The Error Monad Transformer (3)

Any monad transformed by $\text{ET}$ is an instance of $\text{E}$:

$$\text{instance Monad } m \Rightarrow \text{E} \ (\text{ET} \ m) \text{ where}$$

- $\text{eFail} = \text{ET} \ (\text{return Nothing})$
- $m_1 \ 'eHandle' \ m_2 = \text{ET} \ \$$
  ma <- unET m1
  case ma of
    Nothing -> unET m2
    Just _  -> return ma

Exercise 5: Running transf. monads

Let

$$\text{ex1} = \text{eFail} \ 'eHandle' \ \text{return} \ 1$$

1. Suggest a possible type for $\text{ex1}$.
2. How can $\text{ex1}$ be run, given your type?

The Error Monad Transformer (4)

A state monad transformed by $\text{ET}$ is a state monad:

$$\text{instance S } m \ s \Rightarrow \text{S} \ (\text{ET} \ m) \ s \text{ where}$$

- $s\text{Set} \ s = \text{lift} \ (s\text{Set} \ s)$
- $s\text{Get} = \text{lift} \ s\text{Get}$

Exercise 5: Solution

$$\text{ex1} :: \text{ET I Int}$$
$$\text{ex1} = \text{eFail} \ 'eHandle' \ \text{return} \ 1$$

$$\text{ex1r} :: \text{Int}$$
$$\text{ex1r} = \text{runI} \ (\text{runET} \ \text{ex1})$$
The State Monad Transformer (1)

```haskell
newtype ST s m a = ST (s -> m (a, s))
unST (ST m) = m

instance Monad m => Monad (ST s m) where
  return a = ST (
    s -> return (a, s)
  )
  m >>= f = ST $ \s -> do
    (a, s') <- unST m s
    unST (f a) s'
```

The State Monad Transformer (2)

We need the ability to run transformed monads:

```haskell
runST :: Monad m => ST s m a -> s -> m arunST stf s0 = do
  (a, _) <- unST stf s0
  return a
```

ST is a monad transformer:

```haskell
instance Monad m => MonadTransformer (ST s) m where
  lift m = ST (\s -> m >>= \a ->
    return (a, s))
```

The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

```haskell
instance Monad m => S (ST s m) s where
  sSet s = ST (\_ -> return ((), s))
  sGet = ST (\s -> return (s, s))
```

An error monad transformed by ST is an error monad:

```haskell
instance E m => E (ST s m) where
  eFail = lift eFail
  m1 `eHandle` m2 = ST $ \s ->
    unST m1 s `eHandle` unST m2 s
```

Exercise 6: Effect ordering

Consider the code fragment

```haskell
ex2a :: ST Int (ET I) Int
ex2a = (sSet 3 >> eFail) `eHandle` sGet
```

Note that the exact same code fragment also can be typed as follows:

```haskell
ex2b :: ET (ST Int I) Int
ex2b = (sSet 42 >> eFail) `eHandle` sGet
```

What is

```haskell
runI (runET (runST ex2a 0))
runI (runST (runET ex2b 0))
```
Exercise 6: Solution

\[
\begin{align*}
\text{runI } (\text{runET } (\text{runST ex2a } 0)) &= 0 \\
\text{runI } (\text{runST } (\text{runET ex2b}) 0) &= 3
\end{align*}
\]

Exercise 7: Alternative ST?

To think about.

Could \texttt{ST} have been defined in some other way, e.g.

\[
\text{newtype ST s m a} = \text{ST } (m (s \to (a, s)))
\]

or perhaps

\[
\text{newtype ST s m a} = \text{ST } (s \to (m a, s))
\]

Reading (1)


Reading (2)

- Nomaware. \textit{All About Monads}.
  
  \text{http://www.nomaware.com/monads}