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 $M A$ 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)

- 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

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

eval :: Exp -> IntegersafeEval (Lit n) = Just n
safeEval (Add e1 e2) = case safeEval e1 of
  Nothing -> NothingJust n1 ->
  case safeEval e2 of
  Nothing -> NothingJust 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 (or computations).
- If one evaluation fails, fail overall.
- Otherwise, make result available to following evaluations.

Example: Numbering trees

```haskell
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
    in (Node nt1' nt2', 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)

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

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)))
```

Sequencing evaluations (3)

```haskell
safeEval (Mul e1 e2) =
  safeEval e1 `evalSeq` (\n1 -> safeEval e2 `evalSeq` (\n2 -> Just (n1 * n2))
```

Aside: Scope rules of \(\lambda\)-abstractions

The scope rules of \(\lambda\)-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))
```

Inlining `evalSeq` (2)

```haskell
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 n2 -> (Just n1 + n2)
```

Excercise 1: Verify the other cases.

Inlining `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 `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
```

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`.

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 =
```

Sequencing of stateful computations:

```haskell
sSeq :: S a -> (a -> S b) -> S b
sSeq sa f =
```
Stateful Computations (4)

Reading and incrementing the state:

\[
\text{Inc} :: S \text{ Int} \\
\text{Inc} = \n \rightarrow (n, n + 1)
\]

Numbering trees revisited

```hs
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
  \[
  M :: * \rightarrow *
  \]
  \[
  M T
  \]
  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.

Monad laws

Additionally, the following laws must be satisfied:

\[
\text{return } x \gg= f = f x
\]
\[
M \gg= \text{return } = M
\]
\[
(M \gg= f) \gg= g = M \gg= (\lambda x \rightarrow f x \gg= g)
\]
I.e., \text{return} is the right and left identity for \( \gg= \), and \( \gg= \) is associative.

Exercise 2: Solution

```hs
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)

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

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 \text{return} and \( \gg= \).
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.:
return x >>= f = id x >>= f
= x >>= f
= f x

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

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
ml `catch` m2 = case ml of
  Just _ -> ml
  Nothing -> m2

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

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 has two further methods with default instances:
  (>>) :: m a -> m b -> m b
  m >> k = m >>= \

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

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 (
  m >>= f = S $ 

Typical operations on a state monad:
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.:
r m = fst (unS m 0)

The do-notation (1)
Haskell provides convenient syntax for programing 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:

\[
do 
  \text{exp}_1 \\
  \text{exp}_2 \\
  \text{return} \ \text{exp}_3
\]

is syntactic sugar for

\[
\text{exp}_1 \gg= \lambda \rightarrow \\
\text{exp}_2 \gg= \lambda \rightarrow \\
\text{return} \ \text{exp}_3
\]

Nondeterminism: The list monad

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

Example:

\[
do 
  x \leftarrow [1, 2] \\
y \leftarrow ['a', 'b'] \\
return (x,y)
\]

Result: \{\{1,'a'\},\{1,'b'\},\{2,'a'\},\{2,'b'\}\}

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))

The do-notation (3)

A let-construct is also provided:

\[
do 
  \text{let} \ \text{a} = \text{exp}_1 \\
  \text{b} = \text{exp}_2 \\
  \text{return} \ \text{exp}_3
\]

is equivalent to

\[
do 
  \text{a} \leftarrow \text{return} \ \text{exp}_1 \\
  \text{b} \leftarrow \text{return} \ \text{exp}_2 \\
  \text{return} \ \text{exp}_3
\]

Environments: The reader monad

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

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

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

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 :: IO String

Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been
  newtype SE s a = SE (s -> 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.
A monad transformer maps monads to monads. This is represented by a type constructor of the following kind:

\[ T \colon (\, \ast \to \ast \, ) \to (\, \ast \to \ast \, ) \]

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:

\[ \text{lift} \colon M \, a \to T \, M \, a \]

### 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:

\[ \text{class } \text{Monad } m \Rightarrow E \, m \text{ where} \]

\[ \text{eFail} \colon m \, a \to m \, a \]

\[ \text{eHandle} \colon m \, a \to m \, a \to m \, a \]

\[ \text{class } \text{Monad } m \Rightarrow S \, m \, s \mid m \to s \text{ where} \]

\[ \text{sSet} \colon s \to m \, () \]

\[ \text{sGet} \colon m \, s \]

### The Identity Monad

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

newtype I \, a = I \, a
unI \, (I \, a) = a

instance Monad I where

\[ \text{return} \, a = I \, a \]

\[ m >>= f = f \, \text{(unI} \, m) \]

runI :: I \, a \to a
runI = unI

### The Error Monad Transformer (1)

newtype ET \, m \, a = ET \, (m \, (\, \text{Maybe} \, a \, ) )
unET \, (ET \, m) = m

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

\[ \text{return} \, a = ET \, (\, \text{return} \, (\, \text{Just} \, a \, ) \, ) \]

\[ m >>= f = ET \, \$ \, \text{do} \]

\[ ma \leftarrow \text{unET} \, m \]

\[ \text{case} \, ma \, \text{of} \]

\[ \text{Nothing} \to \text{return} \, \text{Nothing} \]

\[ \text{Just} \, a \to \text{unET} \, \{ f \, a \} \]

### The Error Monad Transformer (2)

We need the ability to run transformed monads:

runET :: Monad m \Rightarrow ET \, m \, a \to m \, a
runET \, etm = \text{do} \]

\[ ma \leftarrow \text{unET} \, \text{etm} \]

\[ \text{case} \, ma \, \text{of} \]

\[ \text{Just} \, a \to \text{return} \, a \]

ET is a monad transformer:

instance Monad m \Rightarrow \text{MonadTransformer} \, ET \, m \, where

\[ \text{lift} \, m = ET \, \{ m >>= \, \lambda \, a \to \text{return} \, (\, \text{Just} \, a \, ) \} \]

### The Error Monad Transformer (3)

Any monad transformed by ET is an instance of E:

instance Monad m \Rightarrow E \, (ET \, m) \, where

\[ \text{eFail} = ET \, \{ \text{return} \, \text{Nothing} \} \]

\[ \text{m1} \, \text{eHandle} \, \text{m2} = ET \, \{ \text{do} \]

\[ ma \leftarrow \text{unET} \, m1 \]

\[ \text{case} \, ma \, \text{of} \]

\[ \text{Nothing} \to \text{unET} \, m2 \]

\[ \text{Just} \, a \to \text{return} \, ma \]

### The Error Monad Transformer (4)

A state monad transformed by ET is a state monad:

instance S \, m \, s \Rightarrow S \, (ET \, m) \, s \, where

\[ \text{sSet} \, s = \text{lift} \, \{ \text{sSet} \, s \} \]

\[ \text{sGet} = \text{lift} \, \{ \text{sGet} \} \]

### Exercise 5: Running transf. monads

Let

\[ \text{ex1} = \text{eFail} \, \text{eHandle} \text{'} \, \text{return} \, 1 \]

1. Suggest a possible type for \text{ex1}.
2. How can \text{ex1} be run, given your type?
Exercise 5: Solution

```haskell
ex1 :: ET I Int
ex1 = eFail `eHandle` return 1

ex1r :: Int
ex1r = runI (runET ex1)
```

Exercise 6: Effect ordering

Consider the code fragment

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

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

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

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

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 7: Alternative ST?

To think about.

Could ST have been defined in some other way, e.g.

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

or perhaps

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

Reading (1)


Reading (2)