AN INTRODUCTION TO MONADS
Kathleen Fisher

Reading: “A history of Haskell: Being lazy with class”,
Section 6.4 and Section 7
“Monads for Functional programming”
Sections 13
“Real World Haskell”, Chapter 14: Monads

Thanks to Andrew Tolmach and Simon Peyton Jones for some of these slides.

Reviewing IO Monad
- Basic actions in IO monad have “side effects”:
  
  ```hs
  getChar :: IO Char
  putChar :: Char -> IO ()
  isEOF :: IO Bool
  ```

- “Do” combines actions into larger actions:
  
  ```hs
  echo :: IO ()
  echo = do { b <- isEOF;
              if not b then do
                          x <- getChar; putChar x; echo
              else return () }
  ```

- Operations happen only at the “top level” where we implicitly perform an operation with type
  
  ```hs
  runIO :: IO a -> a
  -- Doesn’t really exist
  ```

More about “do”
- Actions of type IO() don’t carry a useful value, so we can sequence them with >>.
  
  ```hs
  (>>) :: IO a -> IO b -> IO b
  x >>= y = x >>= (\_. -> y)
  ```

- The full translation for “do” notation is:
  
  ```hs
  do { x<e; es } = e >>= \x -> do { es }
  do { e; es } = e >>= do { es }
  do { e } = e
  do {let ds; es} = let ds in do {es}
  ```

```

Notes on the Reading

- “Monads for functional programming” uses
  - `unit` instead of `return`
  - `★` instead of `>>=`
  But it is talking about the same things.

- “Real World Haskell”, Chapter 14, uses running examples introduced in previous chapters. You don’t need to understand all that code, just the big picture.

“do” and “bind”
- The special notation
  
  ```hs
  do {v1 <- e1; e2}
  ```

  is “syntactic” sugar for the ordinary expression
  
  ```hs
  e1 >>= \v1 -> e2
  ```

  where `>>=` (called bind) sequences actions.

  ```hs
  (>>=) :: IO a -> (a -> IO b) -> IO b
  ```

- The value returned by the first action needs to be fed to the second; hence the 2nd arg to `>>=` is a function (often an explicit lambda).

Explicit Data Flow
- Pure functional languages make all data flow explicit.

Advantages
- Value of an expression depends only on its free variables, making equational reasoning valid.
- Order of evaluation is irrelevant, so programs may be evaluated lazily.
- Modularity: everything is explicitly named, so programmer has maximum flexibility.

Disadvantages
- Plumbing, plumbing, plumbing!
An Evaluator

```
data Exp = Plus  Exp Exp
  | Minus Exp Exp
  | Times Exp Exp
  | Div   Exp Exp
  | Const Int

eval :: Exp -> Int
eval (Plus  e1 e2) = (eval e1) + (eval e2)
 eval (Minus e1 e2) = (eval e1) - (eval e2)
 eval (Times e1 e2) = (eval e1) * (eval e2)
 eval (Div   e1 e2) = (eval e1) `div` (eval e2)
 eval (Const i)     = i

answer = eval (Div (Const 3) (Plus (Const 4) (Const 2)))
```

Making Modifications

- **To add error checking**
  - Purely: modify each recursive call to check for and handle errors.
  - Impurely: throw an exception, wrap with a handler.

- **To add logging**
  - Purely: modify each recursive call to thread a log.
  - Impurely: write to a file or global variable.

- **To add a count of the number of operations**
  - Purely: modify each recursive call to thread count.
  - Impurely: increment a global variable.

Clearly the imperative approach is easier!

Adding Error Handling

- **Modify code to check for division by zero:**
  ```
  data Hope a = Ok a | Error String
  eval1 :: Exp -> Hope Int
  eval1 (Div   e1 e2) = case eval1 e1 of
    Ok v1 ->
      case eval1 e2 of
          Ok v2 ->
            if v2 == 0 then Error "divby0"
            else Ok (v1 `div` v2)
          Error s -> Error s
      Error s -> Error s
  eval1 (Const i)     = Ok i
  eval2 :: Exp -> Hope Int
  eval2 (Div   e1 e2) =
    eval2 e1 ifOKthen (v1 ->
      eval2 e2 ifOKthen (v2 ->
        if v2 == 0 then Error "divby0"
        else Ok (v1 `div` v2))
    Error s -> Error s
  eval2 (Const i)     = Ok i
  ```

Add a Useful Abstraction

- We can abstract how Error flows through the code with a higher-order function:

  ```
  ifOKthen :: Hope a -> (a -> Hope b) -> Hope b
  ifOKthen k = case a of Ok x -> k x
                 Error s -> Error s
  eval2 :: Exp -> Hope Int
  eval2 (Times e1 e2) =
    eval2 e1 ifOKthen (v1 ->
      eval2 e2 ifOKthen (v2 ->
        Ok (v1 * v2))
    Error s -> Error s
  eval2 (Div e1 e2) =
    eval2 e1 ifOKthen (v1 ->
      eval2 e2 ifOKthen (v2 ->
        if v2 == 0 then Error "divby0"
        else Ok (v1 `div` v2))
    Error s -> Error s
  eval2 (Const i)     = Ok i
  ```

A Pattern...

- **Compare the types of these functions:**
  ```
  ifOKthen :: Hope a -> (a -> Hope b) -> Hope b
  Ok      :: a -> Hope a
  return :: a -> IO a
  ```

  The similarities are not accidental!

  Like IO, Hope is a monad.

  - IO threads the "world" through functional code.
  - Hope threads whether an error has occurred.

  Monads can describe many kinds of plumbing!
A monad consists of:
- A type constructor M
- A function \( \text{return} :: a \rightarrow M a \)
- A function \( >>= :: M a \rightarrow (a \rightarrow M b) \rightarrow M b \)

Where \( >>= \) and \( \text{return} \) obey these laws:

1. \( \text{return} x >>= k = k x \)
2. \( m >>= \text{return} = m \)
3. \( m >>= (x \rightarrow m1 >>= (y \rightarrow m2 >>= (z \rightarrow m3))) = (m >>= x \rightarrow m1) >>= (y \rightarrow m2) >>= (z \rightarrow m3) \)

\( x \) not in free vars of \( m3 \)

First Monad Law:
\[ \text{return} x >>= k = k x \]

Second Monad Law:
\[ m >>= \text{return} = m \]

Third Monad Law:
\[ m >>= (x \rightarrow m1 >>= (y \rightarrow m2 >>= (z \rightarrow m3))) = (m >>= x \rightarrow m1) >>= (y \rightarrow m2) >>= (z \rightarrow m3) \]

So, there are many different type constructors that are monads, each with these operations—

...that sounds like a job for a type (constructor) class!

We can define type classes over type constructors:

```
class HasMap c where -- HasMap = Functor
  map :: (a->b) -> c a -> c b

instance HasMap [] where
  map f [] = []
  map f (x:xs) = f x : map f xs

instance HasMap Tree where
  map f (Leaf x) = Leaf (f x)
  map f (Node t1,t2) = Node(map f t1, map f t2)

instance HasMap Opt where
  map f (Some s) = Some (f s)
  map f None = None
```

We can do the same thing for monads.

The Haskell Prelude defines a type constructor class for monadic behavior:

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

The Prelude defines an instance of this class for the IO type constructor.

The "do" notation works over any instance of class Monad.
Hope, Revisited

- We can make Hope an instance of Monad:

  ```haskell```
  ```
  instance Monad Hope where
  return = Ok
  (>>=)  = ifOKthen
  ```

- And then rewrite the evaluator to be monadic

  ```haskell```
  ```
  instance Monad Hope where
  return = Ok
  (>>=)  = ifOKthen
  ```

Eval with Monadic Tracing

```haskell```
```
  evalITM :: Exp -> Tr Int
  ```

```
```haskell```
```
  eval13 :: Exp -> Hope Int
  ```

```
```haskell```
```
  eval3 :: Exp -> Hope Int
  ```

```
```haskell```
```
  eval1 :: Exp -> Hope Int
  ```

Adding a Count of Div Ops

- Non-monadically modifying the original evaluator to count the number of divisions requires changes similar to adding tracing:
  - thread an integer count through the code
  - update the count when evaluating a division.

- Monadically, we can use a state monad ST, parameterized over an arbitrary state type. Intuitively:

  ```haskell```
  ```
  type ST a = a -> (a, a)
  ```

  The IO monad can be thought of as an instance of the ST monad, where the type of the state is "World."

  ```haskell```
  ```
  IO = ST World
  ```

```
```haskell```
```
First, we introduce a type constructor for the new monad so we can make it an instance of Monad:

```
newtype s a = ST {runST :: s -> (a,s)}
```

A newtype declaration is just like a datatype, except:
- It must have exactly one constructor.
- Its constructor can have only one argument.
- It describes a strict isomorphism between types.
- It can often be implemented more efficiently than the corresponding datatype.

The curly braces define a record, with a single field named runST with type s -> (a,s).

```
newtype State s a = ST {
  runST :: s -> (a,s),
  return :: a -> ST s a
}
```

We need to make ST an instance of Monad:

```
instance Monad (ST s) where
  return a = ST (s -> (a,s))
  m >>= k = ST (s -> let (a,s') = runST m s in runST (k a) s')
```

The monad structure specifies how to thread the state. Now we need to define operations for using the state.

```
-- Get the value of the state, leave state value unchanged.
get :: ST s s
get = ST (s -> (s,s))

-- Make put's argument the new state, return the unit value.
put :: s -> ST s ()
put s = ST (s -> ((),s))

-- Before update, the state has value s.
-- Return s as value of action and replace s with f s.
update :: (s -> s) -> ST s s
update f = ST (s -> (s,f s))
```

The state flow is specified in the monad; eval can access the state w/o having to thread it explicitly.

```
evalCD :: Exp -> ST Int Int
-- Plus and Minus omitted, but similar.
evalCD (Times e1 e2) = do { v1 <- evalCD e1; v2 <- evalCD e2; return (v1 * v2) }
evalCD (Div e1 e2) = do { v1 <- evalCD e1; v2 <- evalCD e2; update (+1); return (v1 `div` v2) }
evalCD (Const i) = do { return i }
answerCD = runST (evalCD expA) 0
-- (0,1) 0 is the value of expA, 1 is the count of divs.
```

The “Real” ST Monad
The module Control.Monad.ST.Lazy, part of the standard distribution, defines the ST monad, including the get and put functions.

It also provides operations for allocating, writing to, reading from, and modifying named imperative variables in ST s:

```
-- From Data.STRef.Lazy
data STRef s a
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> s -> ST s a
modifySTRef :: STRef s a -> (a -> a) -> ST s a
```

Analogous to the IORefs in the IO Monad.
**Swapping in ST s**

- Using these operations, we can write an imperative swap function:

  ```haskell
  swap :: STRef s a -> STRef s a -> ST s ()
  swap r1 r2 = do {v1 <- readSTRef r1;
                  v2 <- readSTRef r2;
                  writeSTRef r1 v2;
                  writeSTRef r2 v1}
  ```

- And test it...

  ```haskell
  testSwap :: Int
  testSwap = runST (do { r1 <- newSTRef 1;
                          r2 <- newSTRef 2;
                          swap r1 r2;
                          readSTRef r2})
  ```

**But Wait!!!!**

- The analogous function in the IO Monad `unsafePerformIO` breaks the type system.

- How do we know `runST` is safe?

  ```haskell
  let v = runST (newSTRef True)
  in runST (readSTRef v) -- BAD!!
  ```

  This code must be outlawed because actions in different state threads are not sequenced with respect to each other. Purity would be lost.

**A Closer Look**

- Consider again the test code:

  ```haskell
  testSwap :: Int
  testSwap = runST (do { r1 <- newSTRef 1;
                          r2 <- newSTRef 2;
                          swap r1 r2;
                          readSTRef r2})
  ```

  The `runST :: ST s Int -> Int` function allowed us to "escape" the `ST s` monad.

**But How?**

- Initially, the Haskell designers thought they would have to tag each reference with its originating state thread and check each use to ensure compatibility.
  - Expensive, runtime test
  - Obvious implementation strategies made it possible to test the identity of a state thread and therefore break referential transparency.
  - Use the type system!

**A Better Type**

- Intuition: `runST` should only be applied to an ST action which uses `newSTRef` to allocate any references it needs.
  - Or: the argument to `runST` should not make any assumptions about what has already been allocated.
  - Or: `runST` should work regardless of what initial state is given.

- So, its type should be:

  ```haskell
  runST :: \a. (\s. ST s a) -> a
  ```

  which is not a Hindley/Milner type because it has a nested quantifier. It is an example of a rank-2 polymorphic type.

- Precisely typing `runST` solves the problem!

- In Hindley/Milner, the type we have given to `runST` is implicitly universally quantified:

  ```haskell
  runST :: \s.a. (ST s a) -> a
  ```

  But this type isn’t good enough.

**Typing runST**
Consider the example again:

\[ \text{let } v = \text{runST (newSTRef True)} \]

\[ \text{in runST (readSTRef v)} \quad \text{-- Bad!} \]

The type of \( \text{readSTRef } v \) depends upon the type of \( v \), so during type checking, we will discover:

\[ \ldots v : \text{STRef s Bool} \]

To apply \( \text{runST} \) we have to give (\( \text{readSTRef } v \)) the type \( \forall s . \text{ST } s \text{Bool} \).

But the type system prevents this quantifier introduction because \( s \) is in the set of assumptions.

The types don’t match, so a reference cannot escape from a state thread.

In this example, \( v \) is escaping its thread:

During typing, we get:

\[ \text{newSTRef True} :: ST s (\text{STRef } s \text{Bool}) \]

which generalizes to:

\[ \text{newSTRef True} :: \forall s . ST s (\text{STRef } s \text{Bool}) \]

But we still can’t apply \( \text{runST} \). To try, we instantiate its type with \( \text{STRef } s \text{Bool} \) to get:

\[ \text{runST} :: \forall s . (\forall a . ST a s) \rightarrow a \]

\[ \text{runST} :: (\forall a . ST a s) \rightarrow ST a s \]

The types don’t match, so a reference cannot escape from a state thread.

These arguments just give the intuition for why the type preserves soundness.

In 1994, researchers showed the rank-2 type for \( \text{runST} \) makes its use safe.

They used proof techniques for reasoning about polymorphic programs developed by John Mitchell and Albert Meyer.

**Consequence:** we can write functions with pure type that internally use state. The rest of the program cannot tell the difference.

In addition to imperative variables, the ST monad provides mutable arrays with the API:

\[ \text{newArray} :: \text{Ix } i \rightarrow (i, i) \rightarrow \text{elt} \rightarrow ST s (\text{MArray } (s i \text{elt})) \]

\[ \text{readArray} :: \text{Ix } i \rightarrow \text{MArray } (s i \text{elt}) \rightarrow i \rightarrow ST s \text{elt} \]

\[ \text{writeArray} :: \text{Ix } i \rightarrow \text{MArray } (s i \text{elt}) \rightarrow i \rightarrow \text{elt} \rightarrow ST s () \]

**Problem:** Given a graph and a list of “root” vertices, construct a list of trees that form a spanning forest for the graph.

With lazy evaluation, the trees will be constructed on demand, so this construction corresponds to depth-first search.

We can use the ST monad to give a purely functional interface to an imperative implementation of this algorithm.
Imperative Depth First Search

```haskell
defs :: Graph -> [Vertex] -> [Tree Vertex]
defs g vs = runST{
    do { marks <- newArray (bounds g) False;
          where search :: STArray Vertex Bool -> [Vertex] -> ST g [Tree Vertex]
                search [] = return []
                search (vertex:vs) = do {
                        visited <- search marks vertex
                        if visited then
                            search marks vs
                        else
                            do { writeArray marks vertex True;
                                  ts <- search marks (vertex:vs);
                                  vs <- search marks vs;
                                  return (Node vertex ts) } }
          dfs
        } >>-
    return g marks vs } }
```

Quicksort

```haskell
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (> x) xs)
```

Example: Pairs of Factors

```haskell
multiplyTo :: Int -> [(Int,Int)]
multiplyTo n = do { x <- [1..n];
                    y <- [x..n];
                    if (x * y == n) then return (x,y) else bad }
multMult = head (multiplyTo 10)
```

Lazy evaluation ensures that the function produces only as many pairs as the program consumes.

A Monad of Nondeterminism

```haskell
like many other algebraic types, lists form a monad:
instance Monad [] where
    return x = [x]
(a ++ gb) >>= f = f a ++ gb >>= f
```

The bind operator applies the monad to each element of the list. It is often called ‘or Else’ because if the monad is empty, it returns the list it is applied to. It is used extensively with nondeterministic computations, where the bind concatenates the results.

We can view this monad as a representation of nondeterministic computations, where the members of the list are possible outcomes.

With this interpretation, it is useful to define:
```haskell
  unsafe = (+) -- concatenation
  bad = [] -- empty list
```

Example: Eight Queens

```haskell
type Row = Int
type Col = Int
type QPos = (Row,Col)
type Board = [QPos]

safe :: QPos -> [QPos] -> Bool
safe (r,c) (w,e) = r /= w && c /= e && abs(r-w) /= abs(c-e))
pick :: Int -> [Int]
pick 0 = bad
pick n = return n `unsafe` pick (n-1)

add :: [QPos] -> Board -> Board
add q qs | all (safe q) qs = return (q:qs)
         | otherwise = bad

nqueens :: Int -> [Board]
nqueens n = fill_row []
    where fill_row x board | x > n = return board
                           otherwise =
                           do { q <- pick n;
                                board <- add (q,c) board;
                                fill_row (x+1) board; }

queenResult = head (nqueens 8)
```

Example: Eight Queens
Monad Menagerie

- We have seen many example monads
  - IO, Hope (aka Maybe), Trace, ST, Non-determinism
- There are many more...
  - Continuation monad
  - STM: software transactional memory
  - Reader: for reading values from an environment
  - Writer: for recording values (like Trace)
  - Parsers
  - Random data generators (e.g. in Quickcheck)
- Haskell provides many monads in its standard libraries, and users can write more.

Operations on Monads

- In addition to the "do" notation, Haskell leverages type classes to provide generic functions for manipulating monads.

  ```haskell
  -- Convert list of a actions to single [a] action.
  sequence :: Monad m => [m a] -> m [a]
  sequence [] = return []
  sequence (m:ms) = do{ a <- m; as<-sequence ms; return (a:as) }

  -- Apply f to each a, sequence resulting actions.
  mapM :: Monad m => (a -> m b) -> [a] -> m [b]
  mapM f as = sequence (map f as)

  -- "lift" pure function in a monadic one
  liftM :: Monad m => (a -> b) -> m a -> m b
  liftM f ma = ma >>= (f . return)

  -- and the many others in Control.Monad
  ```

Composing Monads

- Given the large number of monads, it is clear that putting them together is useful:
  - An evaluator that checks for errors, traces actions, and counts division operations.
  - They don't compose directly.
- Instead, monad transformers allow us to "stack" monads:
  - Each monad M typically also provides a monad transformer MT that takes a second monad N and adds M actions to N, producing a new monad that does M and N.
- Chapter 18 of RWH discusses monad transformers.

Summary

- Monads are everywhere!
- They hide plumbing, producing code that looks imperative but preserves equational reasoning.
- The "do" notation works for any monad.
- The IO monad allows interactions with the world.
- The ST monad safely allows imperative implementations of pure functions.
- Slogan: Programmable semi-colons. The programmer gets to choose what sequencing means.

A Monadic Skin

- In languages like ML or Java, the fact that the language is in the IO monad is baked in to the language. There is no need to mark anything in the type system because IO is everywhere.
- In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.
- Interesting perspective: It is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.

The Central Challenge

- Notions of Useful
- Arbitrary effects
- Dangerous
- Safe
- No effects
- Useless
Two Basic Approaches: Plan A

- **Default:** No effects
- **Plan:** Selectively permit effects

Types play a major role

- **Two main approaches:**
  - Domain specific languages (SQL, Xquery, Google map/reduce)
  - Wide-spectrum functional languages + controlled effects (e.g. Haskell)

One of Haskell’s most significant contributions is to take purity seriously, and relentlessly pursue Plan B.

Imperative languages will embody growing (and checkable) pure subsets.

--- Simon Peyton Jones