cs242

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 1-3
"Real World Haskell", Chapter 14: Monads

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

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.

Reviewing IO Monad

- Basic actions in IO monad have "side effects":

```
getChar :: IO Char
putChar :: Char -> IO ()
isEOF   :: IO Bool
```

- "Do" combines actions into larger actions:

```
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

```
runIO :: IO a -> a         -- Doesn't really exist
```

"do" and "bind"

- The special notation

```
do {v1 <- e1; e2}
```

  is "syntactic" sugar for the ordinary expression

```
e1 >>= \v1 -> e2
```

  where >>= (called bind) sequences actions.

```
(>>=) :: 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).

More about "do"

- Actions of type `IO()` don't carry a useful value, so we can sequence them with >>.

```
(>>) :: IO a -> IO b -> IO b
e1 >> e2 = e1 >>= (\_ -> e2)
```

- The full translation for "do" notation is:

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}**

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
-- Plus, Minus, Times cases omitted, but similar.
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
```

Yuck!

Adding Error Handling

- Modify code to check for division by zero:

```
data Hope a = Ok a | Error String

eval1 :: Exp -> Hope Int
-- Plus, Minus, Times cases omitted, but similar.
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
```

Note: whenever an expression evaluates to `Error`, that `Error` propagates to final result.

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
e `ifOKthen` k = case e of Ok x -> k x
                           Error s -> Error s
```

```
eval2 :: Exp -> Hope Int
-- Cases for Plus and Minus omitted
eval2 (Times e1 e2) =
    eval2 e1 `ifOKthen` (\v1 ->
    eval2 e2 `ifOKthen` (\v2 ->
    Ok(v1 * v2)))
eval2 (Div   e1 e2) =
    eval2 e1 `ifOKthen` (\v1 ->
    eval2 e2 `ifOKthen` (\v2 ->
    if v2 == 0 then Error "divby0"
              else Ok(v1 `div` v2)))
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   -- constructor for Hope
```

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

Monads, Formally

- A monad consists of:
  - A type constructor M
  - A function `return :: a -> M a`
  - A function `>>= :: M a -> ( a -> M b) -> M b`
- Where `>>=` and `return` obey these laws:

```
(1)  return x >>= k = k x

(2)  m >>= return = m

(3)     m1 >>= (\x->m2 >>= \y->m3)
                    =
       (m1 >>= \x->m2) >>= \y->m3
                  x not in free vars of m3
```

Verifying that Hope is a Monad

```
e `ifOKthen` k = case e of Ok x -> k x
                           Error s -> Error s
```

First Monad Law: `return x >>= k = k x`

```
  Ok x `ifOKthen` k
= case Ok x of Ok x -> k x
               Error s -> Error s
= k x
```

Second Monad Law: `m >>= return = m`

```
  m `ifOKthen`  Ok
= case m of Ok x -> Ok x
            Error s -> Error s
= m
```

Third Monad Law (left as an exercise)

```
m1 >>= (\x->m2 >>= \y->m3) = (m1 >>= \x->m2) >>= \y->m3
```

Many Monads

- A monad consists of:
  - A type constructor M
  - A function `return :: a -> M a`
  - A function `>>= :: M a -> ( a -> M b) -> M b`

So, there are many different type (constructors) that are monads, each with these operations....

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

Recall Type Classes

- We can overload operators to work on many types:

```
(==) :: Int  -> Int  -> Bool
(==) :: Char -> Char -> Bool
(==) :: [Int]-> [Int]-> Bool
```

- Type classes and instances capture this pattern:

```
class Eq a where
  (==) :: a -> a -> Bool  ...

instance Eq Int where
  (==) = primIntEq

instance Eq a => Eq [a] where
  (x:xs) == (y:ys) = x==y && xs == ys
...
```

Recall Type Constructor Classes

- 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 Monad Constructor Class

- 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:

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

- And then rewrite the evaluator to be monadic

```
eval3 :: Exp -> Hope Int
-- Cases for Plus and Minus omitted but similar
eval3 (Times e1 e2) = do {
    v1 <- eval3 e1;
    v2 <- eval3 e2;
    return (v1 * v2)      }
eval3 (Div   e1 e2) =  do {
    v1 <- eval3 e1;
    v2 <- eval3 e2;
    if v2 == 0 then Error "divby0" else return (v1 `div` v2)}
eval3 (Const i)      = return i
```

Compare

```
-- Div case, non-monadic case
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
```

```
-- Div case, monadic case
eval3 (Div   e1 e2) =  do {
    v1 <- eval3 e1;
    v2 <- eval3 e2;
    if v2 == 0 then Error "divby0"
    else return (v1 `div` v2)}
```

The monadic version is much easier to read and modify.

Adding Tracing

- Modify (original) interpreter to generate a log of the operations in the order they are done.

```
evalT :: Exp -> [String] -> ([String], Int)
-- Minus, Times, Div cases omitted, but similar.
evalT (Plus e1 e2) s =
  let (s1,v1) = evalT e1 s
      (s2,v2) = evalT e2 s1
  in  (s2++["+"], v1 + v2)
evalT (Const i) s = (s++[show i], i)

expA = (Div (Const 3)
            (Plus (Const 4) (Const 2)))

(traceTA,answerTA) = evalT expA []
-- (["3","4","2","+","/"],0)
```

More ugly plumbing!

Tracing Monad

- We can capture this idiom with a tracing monad, avoiding having to explicitly thread the log through the computation.

```
data Tr a = Tr [String] a
instance Monad Tr where
  return a = Tr [] a
  m >>= k = let (trace,  a) = runTr m
                (trace', b) = runTr (k a)
            in Tr (trace++trace') b

-- runTr lets us "run" the Trace monad
runTr :: Tr a -> ([String], a)
runTr (Tr s a) = (s,a)

-- trace adds argument to the log
trace :: String -> Tr ()
trace x = Tr [x]  ()
```

Eval with Monadic Tracing

```
evalTM :: Exp -> Tr Int
-- Cases for Plus and Minus omitted but similar
evalTM (Times e1 e2) = do {
    v1 <- evalTM e1;
    v2 <- evalTM e2;
    trace "*";
    return (v1 * v2)      }
evalTM (Div   e1 e2) = do {
    v1 <- evalTM e1;
    v2 <- evalTM e2;
    trace "/";
    return (v1 `div` v2)  }
evalTM (Const i)      = do{trace (show i); return i}

answerTM = runTr (evalTM expA)
-- (["3","4","2","+","/"],0)
```

Which version would be easier to modify?

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:

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

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

```
IO = ST World
type IO a = World -> (a, World)
```

The ST Monad

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

```
newtype State 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)`.
- The name of the field can be used to access the value in the field:

```
runST :: State s a -> s -> (a,s)
```

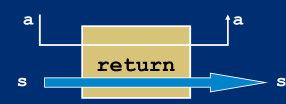
The ST Monad, Continued

- We need to make `ST s` an instance of Monad:

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

```
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')
```

```
return :: a -> ST s a
```



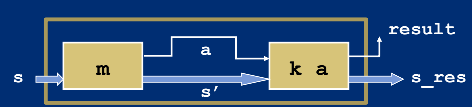
The ST Monad, Continued

- We need to make `ST s` an instance of Monad:

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

```
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')
```

```
>>= :: ST s a -> (a -> ST s b) -> (ST s b)
```



Operations in the ST Monad

- 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))
```

```
-- 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))
```

Counting Divs in the ST Monad

```
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);              -- Increment state by 1.  (\x->x+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 state flow is specified in the monad; eval can access the state w/o having to thread it explicitly.

The "Real" ST Monad

- The module `Control.Module.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 -> a -> ST s ()
modifySTRef :: STRef s a -> (a -> a) -> ST s ()
```

- Analogous to the IORefs in the IO Monad.

Swapping in ST s

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

```
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…

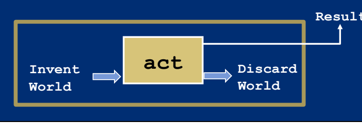
```
testSwap :: Int
testSwap = runST (do { r1 <- newSTRef 1;
                       r2 <- newSTRef 2;
                       swap r1 r2;
                       readSTRef r2})
-- 1
```

A Closer Look

- Consider again the test code:

```
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 Wait!!!!

- The analogous function in the IO Monad `unsafePerformIO` breaks the type system.
- How do we know `runST` is safe?

```
-- What is to prevent examples like this one?
-- It allocates a reference in one state thread,
-- then uses the reference in a different state.

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!

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!

Typing runST

- Precisely typing `runST` solves the problem!
- In Hindley/Milner, the type we have given to `runST` is implicitly universally quantified:

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

- But this type isn't good enough.

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:

```
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.*

How does this work?

- Consider the example again:

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

- The type of `readSTRef v` depends upon the type of `v`, so during type checking, we will discover

```
{...,v:STRef s Bool} |- readSTRef v : ST s Bool
```

- To apply `runST` we have to give `(readSTRef v)` the type `\/s.ST s Bool`.
- But the type system prevents this quantifier introduction because `s` is in the set of assumptions.

A foreign reference cannot be imported into a state thread.

How does this work?

- In this example, v is escaping its thread:

```
v = runST (newSTRef True)              -- Bad!
```

- During typing, we get

```
newSTRef True :: ST s (STRef s Bool)
which generalizes to
newSTRef True :: \/s.ST s (STRef s Bool)
```

- But we still can't apply `runST`. To try, we instantiate its type with `STRef s Bool` to get:

```
runST :: \/s,(\/a.ST s a) -> a          -- instantiate a
runST ::(\/s'.    ST s'(STRef s Bool) -> STRef s Bool
```

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

Formally

- These arguments just give the intuition for why the type preserves soundness.
- In 1994, researchers showed the rank-2 type for `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.

Lazy Functional State Threads by John Launchbury and Simon Peyton Jones

The Implementation

- The ST monad *could be* implemented by threading the state through the computation, directly as the model suggests.
- But, the type system ensures access to state will be single threaded.
- So the system simply does imperative updates.
- The safety of the type system ensures that user code *cannot tell the difference* (except in performance!)

Mutable Arrays

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

```
-- Allocate a new array, with each cell initialized to elt.
newArray   :: Ix i => (i,i) -> elt -> ST s MArray(s i elt)

-- Read an element of the array a[i]
readArray  :: Ix i => MArray(s i elt) -> i -> ST s elt

-- Write an element of the array a[i] := new_elt
writeArray :: Ix i => MArray(s i elt) -> i -> elt -> ST s ()
```

Imperative Depth First Search

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

```
type Graph = Array Vertex [Vertex]
data Tree a = Node a [Tree a]
```

- With lazy evaluation, the trees will be constructed on demand, so the 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.

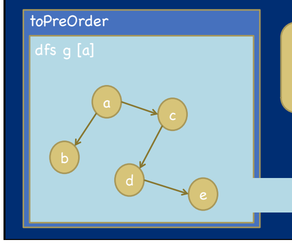
Imperative Depth First Search

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

Using DFS

```
-- Is Vertex b reachable from Vertex a in Graph g?
reachable :: Graph -> Vertex -> Vertex -> Bool
reachable g a b = b `elem` (toPreOrder ( dfs g [a]))

toPreOrder :: [Tree Vertex] -> [Vertex]
```



Lazy evaluation means **e1** will start executing as soon as b is emitted, and dfs will stop, imperative state and all!

```
if reachable g [a] b
then e1
else e2
```

Quicksort

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

The problem with this function is that **it's not really Quicksort**. … What they have in common is overall algorithm: pick a pivot (always the first element), then recursively sort the ones that are smaller, the ones that are bigger, and then stick it all together. But in my opinion the real Quicksort has to be imperative because it relies on destructive update… The partitioning works like this: scan from the left for an element bigger than the pivot, then scan from the right for an element smaller than the pivot, and then swap them. Repeat this until the array has been partitioned.… Haskell has a variety of array types with destructive updates (in different monads), so it's perfectly possible to write the imperative Quicksort in Haskell. [The code is on his blog]

-- Lennart Augustsson

A Monad of Nondeterminism

- Like many other algebraic types, lists form a monad:

```
instance Monad [] where
   return x = [x]
   (x:xs) >>= f = (f x) ++ (xs >>= f)
```

- The bind operator applies `f` to each element `x` in the input list, producing a list for each `x`. Bind then 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:

```
orelse = (++)    -- contatentation
bad = []         -- empty list
```

Example: Pairs of Factors

- This code returns a list of pairs of numbers that multiply to the argument n:

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

fstMult = head (multiplyTo 10)
sndMult = head (tail (multiplyTo 10))
```

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

Example: Eight Queens

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

safe :: QPos -> QPos -> Bool
safe (r,c) (r',c') = r /= r' && c /= c' && (abs(r-r') /= abs(c-c'))

pick :: Int -> [Int]
pick 0 = bad
pick n = return n `orelse` 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 1 []
 where fill_row r board | r > n = return board
                        | otherwise =
                          do { c <- pick n;
                               board' <- add (r,c) board;
                               fill_row (r+1) board';     }

queenResult = head (nqueens 8)
-- [(8,5),(7,7),(6,2),(5,6),(4,3),(3,1),(2,4),(1,8)]
```

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.

```
-- 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

-- 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.

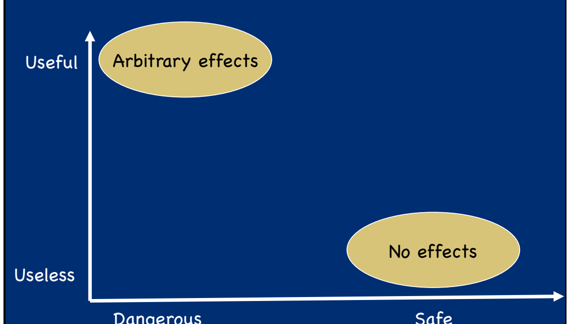
Summary

- 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.

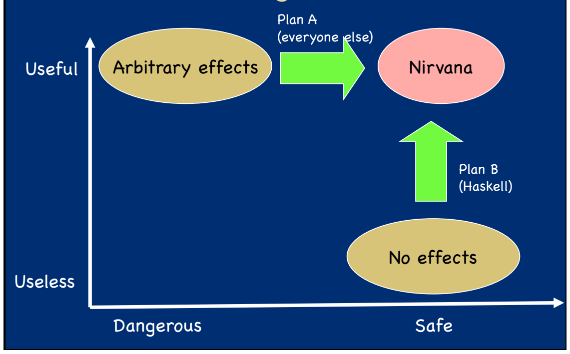
A Monadic Skin

- 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



The Challenge of Effects



Two Basic Approaches: Plan A

Arbitrary effects

Default = Any effect
Plan = Add restrictions

Examples

- Regions
- Ownership types
- Vault, Spec#, Cyclone

Two Basic Approaches: Plan B

Default = No effects
Plan = Selectively permit effects

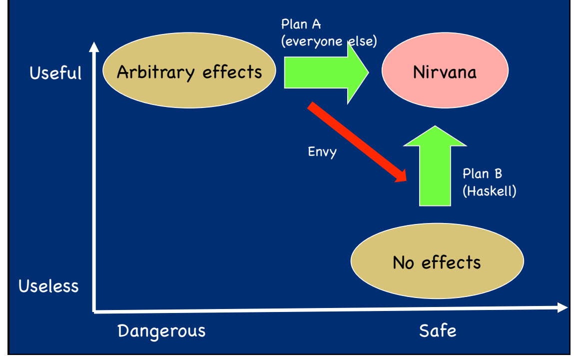
Types play a major role

Two main approaches:

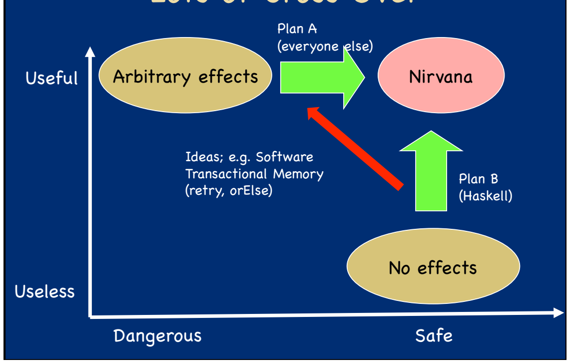
- Domain specific languages (SQL, Xquery, Google map/reduce)
- Wide-spectrum functional languages + controlled effects (e.g. Haskell)

Value oriented programming

Lots of Cross Over



Lots of Cross Over



An Assessment and a Prediction

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