Monads and all that…
II. Monad Transformers

John Hughes

Chalmers University/Quviq AB

Monads from last time...

- State s, for state transformers

```
newtype State s a = State (s -> (a,s))
```

- Maybe, for computations that may fail

```
data Maybe a = Nothing | Just a
```

- Lists (in the exercises), for multiple values

```
data [] a     = []          | (:) a ([] a)
```

- Note the strong similarity Maybe <-> lists!
- Lists ~ Maybe + backtracking

Maybe ~ Lists

- Both provide a way to *fail*
- Both offer a way to *combine alternatives*
  - (i.e. to handle failures)
- It makes sense to define a common interface

MonadPlus

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> ma
```

The same as list append (++)

```
instance MonadPlus Maybe
where
  mzero = Nothing

  Nothing `mplus` m =
    m
  Just a  `mplus` _ =
    Just a
```

```
instance MonadPlus []
where
  mzero = []

  [] `mplus` m =
    m
  (a:as)  `mplus` bs =
    a : (as `mplus` bs)
```

keep the alternatives

N-Queens in the list monad

```
queens 0 = return []
queens n =
  do qs <- queens (n-1)
     q  <- foldr1 mplus (map return [1..8])
     guard (safe q qs)
     return (q:qs)
```

```
return 1 `mplus` return 2
`mplus` return 3…
```

```
*Queens> queens 8 :: [[Integer]]
[[4,2,7,3,6,8,5,1],[5,2,4,7,3,8,6,1],…

*Queens> queens 8 :: Maybe [Integer]
Nothing
```

Let’s write a backtracking parser…

- Parsers need to backtrack
  - use list ☺
- Parsers need to *consume input*
  - use State String ☺
- But we need both at once…
  - hmm…

Putting State and list together

- `State String [a]`

  `s -> ([a],s)`

- `[State String a]`

  `[s -> (a,s)]`

- `s -> [(a,s)]`

  Not a combination of State and something else!

Monads do not compose!

- Given monads m1 and m2,

```
newtype Compose m1 m2 a = Comp (m1 (m2 a))
```

is not a monad!

- Try defining **(>>=)** —you'll fail.

What can we do instead?

- We know what we want:
  - **`s -> [(a,s)]`**
- This *does* use the *list* monad
- Let's *parameterize* the **`State`** monad on an "underlying effect"

```
newtype StateT s m a =
  StateT {runStateT :: s -> m (a,s)}
```

Is this really a monad?

```
newtype StateT s m a =
  StateT {runStateT :: s -> m (a,s)}
```

```
instance Monad m => Monad (StateT s m)
where
  return a = StateT (\s -> return (a,s))
  m >>= f = StateT (\s -> do
            (a,s') <- runStateT m s
            runStateT (f a) s')
```

- StateT s is a *monad transformer*

”Monad Transformer” sounds harder than ”Monad”

BUT IT’S NOT!

- It’s just an easy way to build the monad you want

Monad Transformers

- A monad parameterized on an underlying monad…
- …such that underlying computations can be ”lifted” into the new monad

```
instance MonadTrans t where
  lift :: Monad m => m a -> t m a
```

```
instance MonadTrans (StateT s) where
  lift m = StateT (\s -> do
                     a <- m
                     return (a,s))
```

Can we backtrack and fail?

- i.e. can we define a **MonadPlus** instance, if the underlying monad has one?

```
instance MonadPlus m =>
           MonadPlus (StateT s m) where
  mzero = lift mzero
  m `mplus` m' =
    StateT (\s ->
      runStateT m s `mplus` runStateT m' s)
```

Good for backtracking, Less so for exceptions → both alternatives run in the same state

Get and Put

- **`get`** and **`put`** are easy to implement:

```
get   = StateT (\s  -> return (s,s))
put s = StateT (\s' -> return ((),s))
```

- But now we need **`put`** & **`get`** for **`State s`**, and for **`StateT s m`**
- Define a *class* of monads with state:

```
class Monad m => MonadState s m | m -> s
where
  get :: m s
  put :: s -> m ()
```

So what have we got?

- A *class* defining a set of features we want
  - **`MonadState`**
- A *monad transformer* that adds those features to any monad
  - **`StateT s`**
- Instances of *other* classes, promoting other features from underlying to transformed monad
  - **`MonadPlus`**

Can we do this for other monads?

- Class: **`MonadPlus`**
- Monad transformer: **`MaybeT`**, adding this

```
newtype MaybeT m a =
  MaybeT {runMaybeT :: m (Maybe a)}

instance MonadTrans MaybeT where
  lift m = MaybeT (liftM Just m)
```

```
instance Monad m => Monad (MaybeT m) where
  return x = MaybeT (return (Just x))
  m >>= f = MaybeT (do ma <- runMaybeT m
                       case ma of
                         Nothing -> Nothing
                         Just a -> runMaybeT (f a))
```

Are we adding MonadPlus?

```
instance Monad m => MonadPlus (MaybeT m) where
  mzero = MaybeT (return Nothing)
  m `mplus` m' = MaybeT (do
    ma <- runMaybeT m
    case ma of Nothing -> runMaybeT m'
               Just a  -> return (Just a))
```

Makes it plain that m' sees the underlying effects of m. Good for exception handling, less so for backtracking.

…plus instances to lift other features

- Here's the instance to lift **`State`** operations to **`MaybeT`**:

```
instance MonadState s m => MonadState s (MaybeT m)
where
  get   = lift get
  put s = lift (put s)
```

- A library of *n* monad transformers needs n^2 instance declarations—OK if *n* is not too large

Noncommutativity

```
do put 1
   ((do put 2
        mzero)
     `mplus`
     get)
```

- Returns 1 in StateT Maybe
- Returns 2 in MaybeT State

Monad Transformers Compose!

- If `m` is a monad, so are
  - `StateT s (MaybeT m) a`
  - `MaybeT (StateT s m) a`
  - `StateT s1 (StateT s2 m) a`
  - …
- Given the *identity monad* `Identity`,
  - `MaybeT Identity a   ~ Maybe a`
  - `StateT s Identity a ~ State s a`
  - …

The Identity Monad

- The monad with no features!

```
newtype Identity a =
  Identity {runIdentity :: a}

instance Monad Identity where
  return = Identity
  m >>= f = f (runIdentity m)
```

A Parsing Library

- We want to *add* a state (the input) to the list monad…

```
newtype Parser t a = Parser (StateT [t] [] a)
  deriving (Monad, MonadState [t], MonadPlus)
```

- We also need a function to *run* parsers

```
runParser (Parser m) ts = runStateT m ts
```

Using all this to parse a token

- We can freely combine state & failure ops

```
token :: Parser tok tok
token = do toks <- get
           case toks of
             [] -> mzero
             (t:toks') -> do put toks'
                             return t
```

- Accepting a token satisfying a predicate

```
satisfy p = do t <- token
               guard (p t)
               return t
```

Repetition

Lazy evaluation is critical!

- Operations to repeat a parse 0+ or 1+ times

```
many p = some p `mplus` return []
some p = liftM2 (:) p (many p)
```

- Using them to parse (positive) integers

```
number :: Parser Char Integer
number = do ds <- some (satisfy isDigit)
            return (read ds)
```

Converts a string to the value it denotes

An Arithmetic Expression Parser

```
expr = do a <- term
          exactly '+'
          b <- term
          return (a+b)
       `mplus`
       term

term = do a <- factor
          exactly '*'
          b <- factor
          return (a*b)
       `mplus`
       factor
```

```
factor = number
        `mplus`
         do exactly '('
            a <- expr
            exactly ')'
            return a

exactly t =
  satisfy (==t)
```

Testing the Parser

- Parse the string ”1+2*3”:

```
*Parser> runParser expr "1+2*3"
[(7,""),(3,"*3"),(1,"+2*3")]
```

- Note there are multiple results!
- If we just change **`[]`** to **`Maybe`** in defn of **`Parser`**…

```
*Parser> runParser expr "1+2*3"
Just (7,"")
```

Big Picture

- We have defined a very nice *domain specific language* for backtracking parsers
  - alternation (mplus), repetition (many, some), actions (**do** and return)
- Most of the work was done by the monad transformer *library*
  - the code *specific* to parsing is very short

Hmm…

- list is a *backtracking monad*

- We added *state* to list

- Can we add *backtracking* to an arbitrary monad?

A Backtracking Monad Transformer?

- MaybeT almost does it…

```
newtype MaybeT m a =
  MaybeT {runMaybeT :: m (Maybe a)}
```

Can fail, by producing Nothing, but on success we have no more than one value

- How about

```
newtype BackT m a =
  BackT {unBackT :: MaybeT m (a, BackT m a)}
```

Like a list in which we m-compute each element

Is it a Monad Transformer?

- Lifted operations don’t backtrack:

```
instance MonadTrans BackT where
  lift m = BackT (do a <- lift m
                     return (a,mzero))
```

- **(>>=)** can backtrack into first argument:

```
instance Monad m => Monad (BackT m) where
  return a = lift (return a)
  x >>= f = BackT (do
    (a,back) <- unBackT x
    unBackT (f a `mplus` (back >>= f)))
```

Is it a MonadPlus?

```
instance Monad m => MonadPlus (BackT m)
where
  mzero = BackT mzero
  x `mplus` y =
    BackT (do (a,back) <- unBackT x
              return (a,back `mplus` y)
           `mplus`
           unBackT y)
```

- **`mplus`** is just like list append, **(a,back)** is like **a:back**

Example

```
*BackT> runBackT (return 1 `mplus` return 2)
[1,2]
```

Computes a list of all results in the underlying monad, in this case IO

Now we have a backtracking monad transformer, what shall we do with it?

Let’s implement Prolog!

- Sample Prolog definition:

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

- Prolog defines *predicates*
  – append(Xs,Ys,Zs) is true ⇔ Xs++Ys == Zs
- Prolog execution *solves for unknowns*

Example

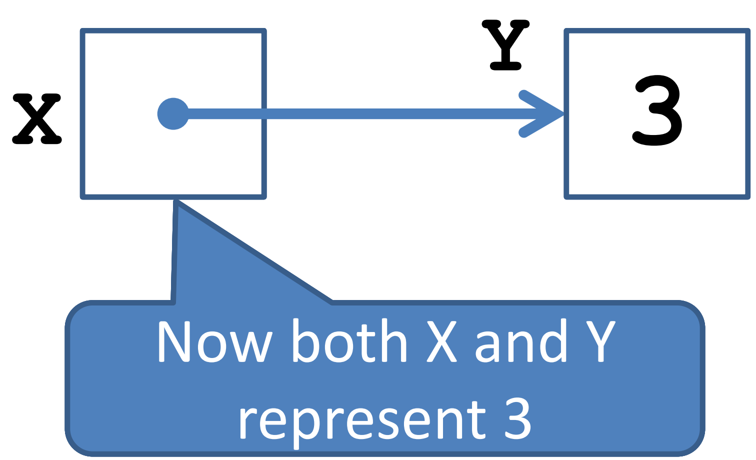
```
?- append([1,2],[3,4],Zs).
Zs = [1,2,3,4]

?- append(Xs,Ys,[1,2,3]).
Xs = [], Ys = [1,2,3];
Xs = [1], Ys = [2,3];
Xs = [1,2], Ys = [3];
Xs = [1,2,3], Ys = []
```

- Prolog finds *multiple solutions* by *backtracking*—multiple clauses may match
- Reverse execution!

What is a Prolog variable?

- A *placeholder* for a value
  - *Single-valued* within one solution
  - Can take *different* values in different solutions

- In the implementation, a cell:



Representing Prolog Values

```
data Logical a = Value a | Var (LogicVar a)

type LogicVar a = IORef (Maybe (Logical a))
```

- What's IORef? Haskell's updateable references

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

We'll work in BackT IO

Lifting IO operations

- We need IO in many situations

```
class Monad m => MonadIO m where
  liftIO :: IO a -> m a
```

- And of course, we can lift it through **BackT** (or any other monad transformer)

```
instance MonadIO m => MonadIO (BackT m)
where
  liftIO io = lift (liftIO io)
```

Our Prolog Monad

```
newtype Logic a = Logic (BackT IO a)
  deriving (Monad, MonadIO, MonadPlus)
```

- We have sequencing, IO, and backtracking… just like that!
- We have to *add* operations on logical variables

Creating a Logical Variable

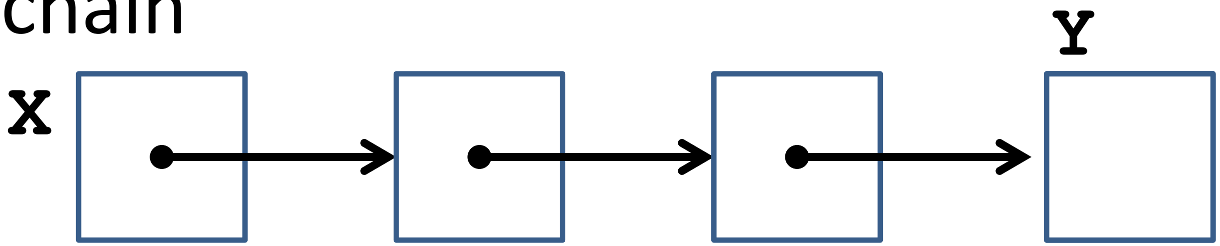
```
data Logical a = Value a | Var (LogicVar a)
type LogicVar a = IORef (Maybe (Logical a))
```

- Variables are created with no contents

```
variable :: Logic (Logical a)
variable = liftM Var (liftIO (newIORef Nothing))
```

Following variable chains

- A variable is always equivalent to the *end* of the chain

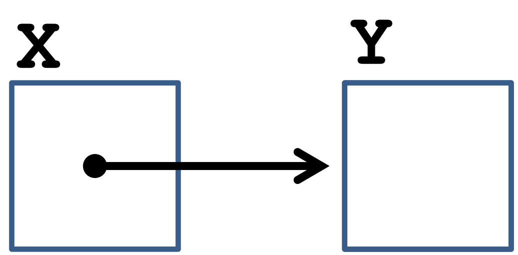



```
data Logical a = Value a | Var (LogicVar a)
type LogicVar a = IORef (Maybe (Logical a))
```

```
follow :: Logical a -> Logic (Logical a)
follow (Value a) = return (Value a)
follow (Var r)   = do
  v <- liftIO (readIORef r)
  case v of
    Nothing -> return (Var r)
    Just val -> follow val
```

Unification

- Variables are assigned values by *unification*
  - e.g. unifying **`[1,2,3]`** with **`[X|Xs]`** assigns **`X=1`**, **`Xs=[2,3]`**
  - Like pattern-matching, except variables may occur on *both* sides
  - Unifying **`X`** and **`Y`**, both unbound, results in



A Unifiable Class

- We'll want to unify all kinds of data…

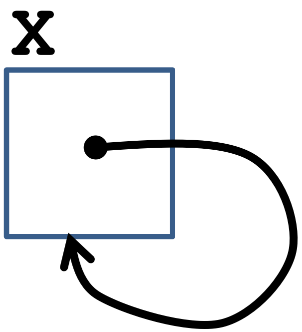
```
class Unifiable a where
  unify :: a -> a -> Logic ()
```

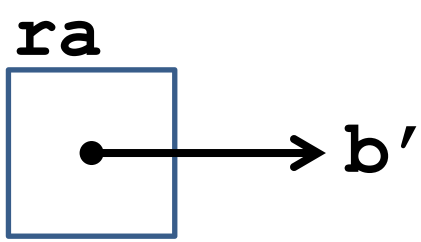
- Unification *makes its arguments equal* by instantiating logical variables—or fails

```
instance Unifiable Integer where
  unify a b = guard (a==b)
```

Unifying variables

```
instance Unifiable a => Unifiable (Logical a) where
  unify a b = do
    a' <- follow a
    b' <- follow b
    case (a',b') of
      (Var ra,Var rb) | ra==rb -> return ()
      (Var ra,_) -> instantiate ra b'
      (_,Var rb) -> instantiate rb a'
      (Value av,Value bv) -> unify av bv
```





Instantiating a variable

- We just write to it

```
instantiate r v =
  liftIO (writeIORef r (Just v))
  `mplus`
  do liftIO (writeIORef r Nothing)
     mzero
```

- But what happens *if we backtrack?*
- We clear variables on backtracking—usually implemented via the ”trail”

Prolog Lists

- Prolog data structures may contain variables at each component

```
data List a = Nil | Cons a (Logical (List a))
```

- And they must be unifiable

```
instance Unifiable a => Unifiable (List a) where
  unify Nil Nil = return ()
  unify (Cons x xs) (Cons y ys) = do
    unify x y
    unify xs ys
  unify _ _ = mzero
```

Let's write Prolog in Haskell!

- Prolog:

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

- Haskell:

```
appendL xs ys zs =
  do unify xs (Value Nil)
     unify ys zs
  `mplus`
  do x   <- variable
     xs' <- variable
     zs' <- variable
     unify xs (Value (Cons x xs'))
     unify zs (Value (Cons x zs'))
     appendL xs' ys zs'
```

Wrapping a test

```
test :: [Integer] -> Logic ([Integer],[Integer])
test zs = do
  xs <- variable
  ys <- variable
  appendL xs ys (toLogical zs)
  liftM2 (,) (fromLogical xs) (fromLogical ys)
```

- Finally:

```
*BackT> runLogic (test [1,2,3])
[([],[1,2,3]),([1],[2,3]),([1,2],[3]),([1,2,3],[])]
```

Conclusion

- Monad transformers make it easy to construct a wide variety of monads
- We can build DSLs with many kinds of effects
- The monad transformer library does a large share of the work