### Monads in Haskell

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

```haskell
class Monad m where
  -- return :: a -> m a
  return = Just

  -- (>>=) :: m a -> (a -> m b) -> m b
  (Just x) >>= f = f x
  Nothing >>= _ = Nothing

  m >>= f = f (m >>= f)
```

### The `Maybe` Monad in Haskell

```haskell
instance Monad Maybe where
  -- return :: a -> Maybe a
  return = Just

  Nothing >>= _ = Nothing
  (Just x) >>= f = f x
```

### Exercise 1: A State Monad in Haskell

Haskell 2010 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 1: Solution

```haskell
instance Monad S where
  return a = S \s -> (a, s)
  m >>= f = f (m >>= f)
```

### 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
m1 `catch` m2 =
  case m1 of
    Just _ -> m1
    Nothing -> m2
```

### Monad-specific Operations (2)

Typical operations on a state monad:

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

get :: S Int
get = S (\s -> (s, s))
```

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

```haskell
runS :: S a -> arunS m = fst (unS m 0)
```

### The `do`-notation (1)

Haskell provides convenient syntax for programming with monads:

```haskell
do
  a <- expr1
  b <- expr2
  return expr3

is syntactic sugar for

```haskell
expr1 >>= \
  \a ->
expr2 >>= \
  \b ->
return expr3
```
The do-notation (2)

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

```haskell
do 
  exp1
  exp2
return exp3
```

is syntactic sugar for

```haskell
exp1 >>= _ ->
exp2 >>= _ ->
return exp3
```

The do-notation (3)

A let-construct is also provided:

```haskell
do 
  let a = exp1 
b = exp2return exp3
```

is equivalent to

```haskell
do 
  a <- return exp1 
b <- return exp2return exp3
```

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')
```

The Compiler Fragment Revisited (1)

Given a suitable “Diagnostics” monad \(D\) that collects error messages, `enterVar` can be turned from this:

```haskell
enterVar :: Id -> Int -> Type -> Env -> Either Env ErrorMsgs
```

into this:

```haskell
enterVarD :: Id -> Int -> Type -> Env -> D Env
```

(Suffix “D” just to remind us the types have changed.)

The Compiler Fragment Revisited (2)

And then `identDefs` is from

```haskell
identDefs :: Int -> Env -> [(Id, Type, Exp ())] -> ([(Id, Type, Exp Attr)], Env, [ErrorMsg])
```

into

```haskell
identDefsD :: Int -> Env -> [(Id, Type, Exp ())] -> D ([(Id, Type, Exp Attr)], Env)
```

with the function definition changing from...

The Compiler Fragment Revisited (3)

```haskell
identDefsD l env [] = return ([], env)
identDefsD l env ((i,t,e) : ds) = do
  e' <- identAuxD l env e
  env' <- enterVarD l i t env
  (ds', env'') <- identDefsD l env' ds
  return ((i,t,e') : ds', env'')
```

The Compiler Fragment Revisited (4)

Compare with the “core” identified earlier!

```haskell
identDefs l env [] = ([], env)
identDefs l env ((i,t,e) : ds) = 
  
  (i,t,e') : ds', env''

where
  e' = identAux l env e 
  env' = enterVar l i t env 
  (ds', env'') = identDefs l env' ds 

The monadic version is very close to this “ideal”, without sacrificing functionality, clarity, or purity!

The List Monad

Computation with many possible results, “nondeterminism”:

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

Example:

```haskell
x <- [1, 2] [(1,'a'), (1,'b') ]
y <- ['a', 'b'] [(2,'a'), (2,'b')] 
return (x,y)
```
The Reader Monad

Computation in an environment:

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

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

The Haskell IO Monad

In Haskell, IO is handled through the IO monad. IO is abstract! Conceptually:

```haskell
newtype IO a = IO (World -> (a, World))
```

Some operations:

- `putChar :: Char -> IO ()`
- `putStrLn :: String -> IO ()`
- `getChar :: IO Char`
- `getLine :: IO String`
- `getContents :: String`

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:

```haskell
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

```haskell
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.

Classes for Specific Effects

A monad transformer adds specific effects to any monad. Thus the effect-specific operations need to be overloaded. For example:

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

Monad Transformers in Haskell (1)

A monad transformer maps monads to monads. Represented by a type constructor `T` of the following kind:

```haskell
T :: ( * -> * ) -> ( * -> * )
```

Additionally, a monad transformer adds computational effects. A mapping `lift` from computations in the underlying monad to computations in the transformed monad is needed:

```haskell
lift :: M a -> T M a
```

Monad Transformers in Haskell (2)

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

```haskell
newtype I a = I a
unI (I a) = a
```

```haskell
instance Monad I where
  return a = I a
  m >>= f = f (unI m)
  runI :: I a -> a
  runI = unI
```
The Error Monad Transformer (1)

newtype ET m a = ET (m (Maybe a))
unET (ET m) = m

Any monad transformed by ET is a monad:

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)

MGS 2012: FUN Lecture 4 – p.28/43

The Error Monad Transformer (2)

We need the ability to run transformed monads:

runET :: Monad m => ET m a -> m a
runET etm = do
    ma <- unET etm
    case ma of
        Nothing -> return Nothing
        Just a -> unET (f a)

ET is a monad transformer:

instance Monad m => MonadTransformer ET m where
    lift m = ET (m >>= \a -> return (Just a))

MGS 2012: FUN Lecture 4 – p.29/43

The Error Monad Transformer (3)

Any monad transformed by ET is an instance of E:

instance Monad m => E (ET m) where
    eFail = ET (return Nothing)
    m1 `eHandle` m2 = ET $ do
        ma <- unET m1
        case ma of
            Nothing -> unET m2
            Just _ -> return ma

MGS 2012: FUN Lecture 4 – p.30/43

The Error Monad Transformer (4)

A state monad transformed by ET is a state monad:

instance S m s => S (ET m) s where
    sSet s = lift (sSet s)
    sGet = lift sGet

Exercise 2: Running Transf. Monads

Let

ex2 = eFail `eHandle` return 1

1. Suggest a possible type for ex2.
   (Assume 1 :: Int.)

2. Given your type, use the appropriate combination of "run functions" to run ex2.

Exercise 2: Solution

ex2 :: ET I Int
ex2 = eFail `eHandle` return 1

ex2result :: Int
ex2result = runI (runET ex2)

MGS 2012: FUN Lecture 4 – p.32/43

The State Monad Transformer (1)

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

Any monad transformed by ST is a monad:

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'

MGS 2012: FUN Lecture 4 – p.34/43

The State Monad Transformer (2)

We need the ability to run transformed monads:

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

ST is a monad transformer:

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

MGS 2012: FUN Lecture 4 – p.35/43

The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

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:

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

MGS 2012: FUN Lecture 4 – p.36/43
Exercise 3: Effect Ordering

Consider the code fragment

```haskell
ex3a :: (ST Int (ET I)) Int
  ex3a = (sSet 42 >> eFail) `eHandle` sGet
```
Note that the exact same code fragment also can be typed as follows:

```haskell
ex3b :: (ET (ST Int I)) Int
  ex3b = (sSet 42 >> eFail) `eHandle` sGet
```
What is

```haskell
runI (runET (runST ex3a 0))
runI (runST (runET ex3b) 0)
```

Exercise 3: Solution (1)

```haskell
runI (runET (runST ex3a 0)) = 0
runI (runST (runET ex3b) 0) = 42
```
Why? Because:

- \( \text{ST } s \ (\text{ET } I) \ a \cong s \to (\text{ET } I) \ (a, s) \)
- \( s \to I \ (\text{Maybe } (a, s)) \)
- \( s \to \text{Maybe } (a, s) \)
- \( \text{ET } (\text{ST } s \ I) \ a \cong (\text{ST } s \ I) \ (\text{Maybe } a) \)
- \( s \to I \ (\text{Maybe } a, s) \)
- \( s \to (\text{Maybe } a, s) \)

Exercise 3: Solution (2)

Note that

- \( \text{ET } (\text{ST } s \ I) \ a \cong s \to (\text{Maybe } a, s) \)
results in a notion of a **shared, global** state, while
- \( \text{ST } s \ (\text{ET } I) \ a \cong s \to \text{Maybe } (a, s) \)
has a **transactional** flavour: only if a computation succeeds will any effects from that computation be taken into account.

Both are natural and useful; hence there is no "right" or "wrong" ordering.

Exercise 4: Alternative \texttt{ST}?

To think about.

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

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

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

Problems with Monad Transformers

- With one transformer for each possible effect, we get a lot of combinations: the number grows quadratically; each has to be instantiated explicitly.
- Jaskelioff (2008,2009) has proposed a possible, more extensible alternative.

Reading (1)


Reading (2)