

MGS 2009: FUN Lecture 4

More about Monads

Henrik Nilsson

University of Nottingham, UK

This Lecture

- Monads in Haskell
- Some standard monads
- Combining effects: monad transformers

Monads in Haskell

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
```

Allows names of the common functions to be overloaded and sharing of derived definitions.

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
```

Exercise 1: 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 1: Solution

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

m >>= f = S $ \s ->
  let (a, s') = unS m s
  in unS (f a) s'
```

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

Monad-specific Operations (2)

Typical operations on a state monad:

```
set :: Int -> S ()
```

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

```
get :: S Int
```

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

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

```
runS :: S a -> a
```

```
runS m = fst (unS m 0)
```


The `do`-notation (1)

Haskell provides convenient syntax for programming with monads:

```
do
  a <- exp1
  b <- exp2
  return exp3
```

is syntactic sugar for

```
exp1 >>= \a ->
exp2 >>= \b ->
return exp3
```

The `do`-notation (2)

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

```
do
  exp1
  exp2
  return exp3
```

is syntactic sugar for

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

The do-notation (3)

A `let`-construct is also provided:

```
do
  let a = exp1
      b = exp2
  return exp3
```

is equivalent to

```
do
  a <- return exp1
  b <- return exp2
  return exp3
```

Numbering Trees in do-notation

```
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 \mathbb{D} that collects error messages, `enterVar` can be turned from this:

```
enterVar :: Id -> Int -> Type -> Env  
         -> Either Env ErrorMgs
```

into this:

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

and then `identDefs` from this ...

The Compiler Fragment Revisited (2)

```
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
  where
    (e', ms1) = identAux l env e
    (env', ms2) =
      case enterVar i l t env of
        Left env' -> (env', [])
        Right m    -> (env, [m])
    (ds', env'', ms3) =
      identDefs l env' ds
```

The Compiler Fragment Revisited (3)

into this:

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

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

The Compiler Fragment Revisited (4)

Compare with the “core” identified earlier!

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

The monadic version is very close to ideal,
without sacrificing functionality, clarity, or
pureness!

The List Monad

Computation with many possible results,
“nondeterminism”

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

Example:

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

Result:

```
[(1, 'a'), (1, 'b'),
 (2, 'a'), (2, 'b')]
```

The Reader Monad

Computation in an environment:

```
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:

```
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 (1)

What if we need to support more than one type of effect?

Monad Transformers (1)

What if we need to support more than one type of effect?

For example: State and Error/Partiality?

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))
```

-
-
-

Monad Transformers (2)

However:

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))
```


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:

Monad Transformers (3)

Monad Transformers can help:

- A **monad transformer** transforms a monad by adding support for an additional effect.

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.

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**.

Monad Transformers in Haskell (1)

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

$$T :: (* -> *) -> (* -> *)$$

Monad Transformers in Haskell (1)

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

$$T :: (* -> *) -> (* -> *)$$

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

$$\text{lift} :: M a -> T M a$$

Monad Transformers in Haskell (2)

- These requirements are captured by the following (multi-parameter) type class:

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


Classes for Specific Effects

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

```
class Monad m => E m where
  eFail :: m a
  eHandle :: m a -> m a -> m a
```

```
class Monad m => S m s | m -> s where
  sSet :: s -> m ()
  sGet :: 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
    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)
```

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
        Just a -> return a
```

ET is a monad transformer:

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

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
```

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)
```


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'
```

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))
```

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
```

Exercise 3: Effect Ordering

Consider the code fragment

```
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:

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

What is

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

Exercise 3: Solution

```
runI (runET (runST ex3a 0)) = 0
```

```
runI (runST (runET ex3b) 0) = 42
```

Why? Because:

```
ST s (ET I) a ≅ s -> (ET I) (a, s)
              ≅ s -> I (Maybe (a, s))
              ≅ s -> Maybe (a, s)
ET (ST s I) a ≅ (ST s I) (Maybe a)
              ≅ s -> I (Maybe a, s)
              ≅ s -> (Maybe a, s)
```

Exercise 4: Alternative `ST`?

To think about.

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

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

or perhaps

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

Reading

- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- Sheng Liang, Paul Hudak, Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, January 1995, San Francisco, California