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
    (>>=) :: 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

```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:

```haskell
defined S a = S (Int -> (a, Int))
```

```haskell
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 = 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:

```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, a)))

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

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

```haskell
runS :: S a -> a
runS m = fst (unS m 0)
The \textbf{do-notation (1)}

Haskell provides convenient syntax for programming with monads:
\begin{verbatim}
do
  a <- exp_1
  b <- exp_2
  return exp_3
\end{verbatim}
is syntactic sugar for
\begin{verbatim}
exp_1 >>= \a ->
exp_2 >>= \b ->
return exp_3
\end{verbatim}

The \textbf{do-notation (2)}

Computations can be done solely for effect, ignoring the computed value:
\begin{verbatim}
do
  exp_1
  exp_2
  return exp_3
\end{verbatim}
is syntactic sugar for
\begin{verbatim}
exp_1 >>= \_ ->
exp_2 >>= \_ ->
return exp_3
\end{verbatim}

The \textbf{do-notation (3)}

A \texttt{let}-construct is also provided:
\begin{verbatim}
do
  let a = exp_1
    b = exp_2
    return exp_3
\end{verbatim}
is equivalent to
\begin{verbatim}
do
  a <- return exp_1
  b <- return exp_2
  return exp_3
\end{verbatim}

\textbf{Numbering Trees in do-notation}

\begin{verbatim}
numberTree :: Tree a -> Tree Int
numberTree t = runS (ntAux t)
where
  ntAux :: Tree a -> S (Tree Int)
  ntAux (Leaf _) = do
    n <- getset (n + 1)
    return (Leaf n)
  ntAux (Node t1 t2) = do
    t1' <- ntAux t1
    t2' <- ntAux t2
    return (Node t1' t2')
\end{verbatim}
Given a suitable “Diagnostics” monad \( D \) that collects error messages, `enterVar` can be turned from this:

\[
\text{enterVar} :: \text{Id} \to \text{Int} \to \text{Type} \to \text{Env} \to \text{Either Env ErrorMgs}
\]

into this:

\[
\text{enterVarD} :: \text{Id} \to \text{Int} \to \text{Type} \to \text{Env} \to D \text{ Env}
\]

and then `identDefs` from this . . .

\[
\text{identDefs} l \text{ env} [] = ([], \text{env}, [])
\]
\[
\text{identDefs} l \text{ env} ((i,t,e) : ds) =
\]
\[
((i,t,e') : ds', \text{env''}, \text{ms}1+\text{ms}2+\text{ms}3)
\]

where
\[
\begin{align*}
(e', \text{ms}1) &= \text{identAux} l \text{ env} e \\
(\text{env'}, \text{ms}2) &= \\
\text{case enterVar i l t env of} \\
\text{Left env'} &\to (\text{env'}, []) \\
\text{Right m} &\to (\text{env, m})
\end{align*}
\]
\[
\text{ds'}, \text{env''}, \text{ms}3) = \\
\text{identDefs} l \text{ env'} \text{ ds}
\]

into this:

\[
\text{identDefsD} l \text{ env} [] = \text{return} ([], \text{env})
\]
\[
\text{identDefsD} l \text{ env} ((i,t,e) : ds) = \text{do}
\]
\[
e' \leftarrow \text{identAuxD} l \text{ env} e \\
\text{env'} \leftarrow \text{enterVarD i l t env} (\text{ds'}, \text{env''}) \\
\text{return} ((i,t,e') : \text{ds'}, \text{env''})
\]
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:

```
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:

```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 ()
putStr :: String -> 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*.

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 $\text{lift}$ 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)

- These requirements are captured by the following (multi-parameter) type class:
  ```haskell
  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 need to be overloaded. For example:

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

```haskell
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:

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

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

Any monad transformed by ET is a monad:

```haskell
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:

```haskell
runET :: Monad m => ET m a -> m a
runET etm = do
    ma <- unET etm
    case ma of
        Just a -> return a
        Nothing -> error "Should not happen"
```

ET is a monad transformer:

```haskell
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:

```haskell
class 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
```

Exercise 2: Running Transf. Monads

Let

```haskell```

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

```haskell```

```haskell```

The Error Monad Transformer (4)

A state monad transformed by ET is a state monad:

```haskell
class S m s => S (ET m) s where
    sSet s = lift (sSet s)
    sGet = lift sGet
```

```haskell```

```haskell```

```haskell```
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 arunST 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 s = 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 eFailm1 `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

\[
\text{runI (runET (runST ex3a 0))} = 0 \\
\text{runI (runST (runET ex3b) 0)} = 42
\]

Why? Because:

\[
\text{ST s (ET I) a} \equiv s \to (\text{ET I}) (a, s) \\
\text{ET (ST s I) a} \equiv (\text{ST s I}) (\text{Maybe a}) \\
\]

Exercise 4: Alternative \text{ST}?

To think about.

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

\[
\text{newtype ST s m a = ST (m (s \to (a, s)))}
\]

or perhaps

\[
\text{newtype ST s m a = ST (s \to (m a, s))}
\]

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)
