MGS 2012: 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*:

```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.
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 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))
```

```haskell
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:

```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:

\[
\begin{align*}
\text{set} & : \text{Int} \to \text{S} () \\
\text{set } a & = \text{S} (\_ \to ((), a))
\end{align*}
\]

\[
\begin{align*}
\text{get} & : \text{S} \text{ Int} \\
\text{get} & = \text{S} (\_s \to (s, s))
\end{align*}
\]

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

\[
\begin{align*}
\text{runS} & : \text{S a} \to a \\
\text{runS } m & = \text{fst} (\text{unS } m 0)
\end{align*}
\]
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')
Given a suitable “Diagnostics” monad $D$ that collects error messages, $\text{enterVar}$ can be turned from this:

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

into this:

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

(Suffix “D” just to remind us the types have changed.)
The Compiler Fragment Revisited (2)

And then `identDefs` 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 ...
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
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'')
The Compiler Fragment Revisited (4)

Compare with the “core” identified earlier!

\[
\text{identDefs} \; l \; \text{env} \; [] = ([], \; \text{env}) \\
\text{identDefs} \; l \; \text{env} \; ((i,t,e) : ds) = \\
((i,t,e') : ds' , \; \text{env''})
\]

where

\[
e' = \text{identAux} \; l \; \text{env} \; e \\
\text{env'} = \text{enterVar} \; i \; l \; t \; \text{env} \\
(ds', \; \text{env''}) = \text{identDefs} \; l \; \text{env'} \; ds
\]

The monadic version is very close to this “ideal”, without sacrificing functionality, clarity, or pureness!
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]
y <- ['a', 'b']
return (x, y)
```

Result:

```
[(1,'a'),(1,'b'),
(2,'a'),(2,'b')]
```
The Reader Monad

Computation in an environment:

\[
\text{instance Monad } ((\to) e) \text{ where } \\
\quad \text{return } a = \text{const } a \\
\quad m \triangleright\triangleright f = \lambda e \to f (m e) e \\
\]

\[
\text{getEnv :: } ((\to) e) e \\
\text{getEnv} = \text{id}
\]
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 :: 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:

\[
\text{newtype } SE \ s \ a = SE (s \rightarrow \text{Maybe } (a, s))
\]
Monad Transformers (2)

However:
 Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been

\[
\text{newtype } \text{SE } s \text{ a } = \text{SE } (s \rightarrow (\text{Maybe } a, s))
\]
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 can help:
Monad Transformers can help:

- A *monad transformer* transforms a monad by adding support for an additional effect.
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 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*. 
A **monad transformer** maps monads to monads. Represented by a type constructor $T$ of the following kind:

$$T :: (\ast \rightarrow \ast) \rightarrow (\ast \rightarrow \ast)$$
A **monad transformer** maps monads to monads. Represented by a type constructor $T$ of the following kind:

$$T : : (\star \rightarrow \star) \rightarrow (\star \rightarrow \star)$$

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

$$lift : : M a \rightarrow T M a$$
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 needs to be overloaded. For example:

```haskell
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:

\[
\text{newtype } I\ a = I\ a \\
\text{unI } (I\ a) = a
\]

\[
\text{instance Monad I where} \\
\text{return } a = I\ a \\
\text{m } >>= f = f\ (\text{unI } m)
\]

\[
\text{runI :: I } a \rightarrow a \\
\text{runI } = \text{unI}
\]
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:

```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))
```
Any monad transformed by \( ET \) is an instance of \( E \):

\[
\text{instance Monad } m \Rightarrow E \ (ET \ m) \text{ where }
\]
\[
e\text{Fail} = ET \ (\text{return Nothing})
\]
\[
m1 \ 'eHandle' \ m2 = ET \ $ \ do
\]
\[
ma \leftarrow \text{unET} \ m1
\]
\[
\text{case } ma \text{ of}
\]
\[
\text{Nothing} \rightarrow \text{unET} \ m2
\]
\[
\text{Just } _- \rightarrow \text{return} \ ma
\]
A state monad transformed by ET is a state monad:

\[
\text{instance } S \ m \ s \Rightarrow S \ (ET \ m) \ s \ \text{where}
\begin{align*}
  \text{sSet} & \ = \ \text{lift} \ (\text{sSet} \ s) \\
  \text{sGet} & \ = \ \text{lift} \ \text{sGet}
\end{align*}
\]
Exercise 2: Running Transf. Monads

Let

\[ \text{ex2} = \text{eFail} \ '\text{eHandle}' \ return \ 1 \]

1. Suggest a possible type for \text{ex2}. (Assume \(1 :: \text{Int}\).)

2. Given your type, use the appropriate combination of “run functions” to run \text{ex2}. 
Exercise 2: Solution

\[
\begin{align*}
ex2 &:: \ ET \ I \ Int \\
ex2 &= eFail \ 'eHandle' \ return \ 1 \\
ex2result &:: \ Int \\
ex2result &= runI \ (runET \ ex2)
\end{align*}
\]
The State Monad Transformer (1)

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

Any monad transformed by \texttt{ST} is a monad:

\begin{verbatim}
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'
\end{verbatim}
We need the ability to run transformed monads:

\[
\text{runST} :: \text{Monad } m \Rightarrow \text{ST } s \text{ m a } \rightarrow s \rightarrow m a
\]

\[
\text{runST} \ \text{stf} \ \text{s0} = \text{do}
\]

\[
(a, \_ ) \leftarrow \text{unST} \ \text{stf} \ \text{s0}
\]

\[
\text{return} \ a
\]

\text{ST is a monad transformer:}

\[
\text{instance Monad } m \Rightarrow
\]

\[
\text{MonadTransformer (ST s) m where}
\]

\[
\text{lift } m = \text{ST} (\ \text{s } \rightarrow \ m \triangleright\triangleright= \ \text{\s} \rightarrow \ m \triangleright\triangleright= \ \text{\a } \rightarrow
\]

\[
\text{return} \ (a, \ s))
\]
Any monad transformed by $\text{ST}$ is an instance of $S$:

```haskell
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 $\text{ST}$ is an error monad:

```haskell
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

```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)

\[
\text{runI \ (runET \ (runST \ ex3a \ 0))} = ??? \\
\text{runI \ (runST \ (runET \ ex3b) \ 0)} = ???
\]
Exercise 3: Solution (1)

\[
\text{runI} \ (\text{runET} \ (\text{runST} \ ex3a \ 0)) = 0
\]
\[
\text{runI} \ (\text{runST} \ (\text{runET} \ ex3b) \ 0) = ???
\]
Exercise 3: Solution (1)

\[
\text{runI } (\text{runET } (\text{runST ex3a } 0)) = 0 \\
\text{runI } (\text{runST } (\text{runET ex3b}) 0) = 42
\]
Exercise 3: Solution (1)

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

Why?
Exercise 3: Solution (1)

runI \left( \text{runET} \left( \text{runST} \text{ ex3a} \ 0 \right) \right) = 0
runI \left( \text{runST} \left( \text{runET} \text{ ex3b} \ 0 \right) \right) = 42

Why? Because:

\begin{align*}
\text{ST} \ s \ \text{(ET I)} \ a & \equiv s \to (\text{ET I}) (a, s) \\
& \equiv s \to I (\text{Maybe} (a, s)) \\
& \equiv s \to \text{Maybe} (a, s) \\
\text{ET} \ (\text{ST} \ s \ \text{I}) \ a & \equiv (\text{ST} \ s \ \text{I}) (\text{Maybe} a) \\
& \equiv s \to I (\text{Maybe} a, s) \\
& \equiv s \to (\text{Maybe} a, s)
\end{align*}
Exercise 3: Solution (2)

Note that

\[ \text{ET} \ (\text{ST} \ s \ I) \ a \cong s \rightarrow \ (\text{Maybe} \ a, \ s) \]

results in a notion of a *shared, global* state, while

\[ \text{ST} \ s \ (\text{ET} \ I) \ a \cong s \rightarrow \ \text{Maybe} \ (a, \ s) \]

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 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 (2)