LiU-FP2016: Lecture 9

Monads in Haskell

Henrik Nilsson

University of Nottingham, UK

This Lecture

- Monads in Haskell
- The Haskell Monad Class Hierarchy
- Some Standard Monads and Library Functions

Monads in Haskell (1)

In Haskell, the notion of a monad is captured by a ***Type Class***. In principle (but not quite from GHC 7.8 onwards):

```
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.

Monads in Haskell (2)

The Haskell monad class has two further methods with default definitions:

```
(>>) :: m a -> m b -> m b
m >> k = m >>= \_ -> k

fail :: String -> m a
fail s = error s
```

(However, `fail` will likely be moved into a separate class `MonadFail` in the future.)

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
```


The Monad Type Class Hierachy (1)

Monads are mathematically related to two other notions:

- Functors
- Applicative Functors

Every monad is an applicative functor, and every applicative functor (and thus monad) is a functor.

Class hierarchy:

```
class Functor f where ...
class Functor f => Applicative f where ...
class Applicative m => Monad m where ...
```


The Monad Type Class Hierachy (2)

For example, `fmap` can in principle be defined in terms of `>>=` and `return`, demonstrating that a monad is a functor:

```
fmap f m = m >>= \x -> return (f x)
```

A consequence of this class hierarchy is that to make some `T` an instance of `Monad`, an instance of `T` for both `Functor` and `Applicative` must also be provided.

Applicative Functors (1)

An applicative functor is a functor with application, providing operations to:

- embed pure expressions (`pure`), and
- sequence computations and combine their results (`<*>`)

satisfying some laws.

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Applicative Functors (2)

- Like monads, applicative functors is a notion of computation.
- The key difference is that the result of one computation is not made available to subsequent computations. As a result, the structure of a computation is static.
- Applicative functors are frequently used in the context of parsing combinators. In fact, that is where their origin lies.

Applicative Functors and Monads

A requirement is `return = pure`.
In fact, the `Monad` class provides a default definition of `return` defined that way:

```
class Functor m => Monad m where
    return :: a -> m a
    return = pure

    (>>=) :: m a -> (a -> m b) -> m b
```


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:

```
newtype S a = S { unS :: (Int -> (a, Int)) }
```

(Thus: `unS :: S a -> (Int -> (a, Int))`)

Provide a `Monad` instance for `S`, ignoring for now that instances for `Functor` and `Applicative` are also needed.

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'
```

The Complete Set of S Instances (1)

```
instance Functor S where
    fmap f sa = S $ \s ->
        let
            (a, s’) = unS sa s
        in
            (f a, s’)
```


The Complete Set of S Instances (2)

```
instance Applicative S where
    pure a = S $ \s -> (a, s)

    sf <*> sa = S $ \s ->
        let
            (f, s’) = unS sf s
        in
            unS (fmap f sa) s’
```


The Complete Set of S Instances (3)

```
instance Monad S where
    m >>= f = S $ \s ->
        let (a, s’) = unS m s
        in unS (f a) s’
```

(Using the default definition `return = pure`.)

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 `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!

Monadic Utility Functions (1)

Some monad utilities:

```
sequence   :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
mapM     :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_    :: Monad m => (a -> m b) -> [a] -> m ()
when     :: Monad m => Bool -> m () -> m ()
foldM    :: Monad m =>
                (a -> b -> m a) -> a -> [b] -> m a
liftM    :: Monad m => (a -> b) -> m a -> m b
liftM2   :: Monad m =>
                (a -> b -> c) -> m a -> m b -> m c
```

(`liftM` = `fmap`; partly historical.)

Monadic Utility Functions (2)

Example: Suppose we’re given a list `xs` of elements of type `T1` to process in some monad `M`:

- Process `xs` effectfully: `proc :: T1 -> M T2`
- Pick “good” results: `good :: T2 -> Bool`
- “Print” a warning if no good results: `print :: String -> M ()`

```
do
    ys <- mapM proc xs
    let gys = filter good ys
    when (null gys) (print "No good!")
    return gys
```


The List Monad

Computation with many possible results, “nondeterminism”:

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

Example:	Result:

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


The ST Monad: “Real” State

The ST monad (common Haskell extension) provides real, imperative state behind the scenes to allow efficient implementation of imperative algorithms:

```
data ST s a -- abstract
instance Monad (ST s)

newSTRef    :: s ST a (STRef s a)
readSTRef   :: STRef s a -> ST s a
writeSTRef  :: STRef s a -> a -> ST s ()

runST :: (forall s . st s a) -> a
```


Reading

- Philip Wadler. The Essence of Functional Programming. *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL’92)*, 1992.
- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.