JFP **13** (1): 199–204, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803002211 Printed in the United Kingdom

Chapter 20

Monad Utilities

```
module Monad (
    MonadPlus(mzero, mplus),
    join, guard, when, unless, ap,
    msum,
    filterM, mapAndUnzipM, zipWithM, zipWithM_, foldM,
    liftM, liftM2, liftM3, liftM4, liftM5,
    -- ...and what the Prelude exports
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    ) where

class  Monad m => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a -> m a -> m a

join              :: Monad m => m (m a) -> m a
guard             :: MonadPlus m => Bool -> m ()
when              :: Monad m => Bool -> m () -> m ()
unless            :: Monad m => Bool -> m () -> m ()
ap                :: Monad m => m (a -> b) -> m a -> m b

mapAndUnzipM      :: Monad m => (a -> m (b,c)) -> [a] -> m ([b], [c])
zipWithM          :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM_         :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
foldM             :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
filterM           :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

```
msum             :: MonadPlus m => [m a] -> m a

liftM            :: Monad m => (a -> b) -> (m a -> m b)
liftM2           :: Monad m => (a -> b -> c) -> (m a -> m b -> m c)
liftM3           :: Monad m => (a -> b -> c -> d) ->
                               (m a -> m b -> m c -> m d)
liftM4           :: Monad m => (a -> b -> c -> d -> e) ->
                               (m a -> m b -> m c -> m d -> m e)
liftM5           :: Monad m => (a -> b -> c -> d -> e -> f) ->
                               (m a -> m b -> m c -> m d -> m e -> m f)
```

The `Monad` library defines the `MonadPlus` class, and provides some useful operations on monads.

20.1 Naming Conventions

The functions in this library use the following naming conventions:

- A postfix "`M`" always stands for a function in the Kleisli category: `m` is added to function results (modulo currying) and nowhere else. So, for example,

```
filter  ::             (a ->   Bool) -> [a] ->   [a]
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

- A postfix "`_`" changes the result type from `(m a)` to `(m ())`. Thus (in the `Prelude`):

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

- A prefix "`m`" generalises an existing function to a monadic form. Thus, for example:

```
sum  :: Num a       => [a]   -> a
msum :: MonadPlus m => [m a] -> m a
```

20.2 Class `MonadPlus`

The `MonadPlus` class is defined as follows:

```
class  Monad m => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a -> m a -> m a
```

The class methods `mzero` and `mplus` are the zero and plus of the monad.

https://doi.org/10.1017/S0956796803002211 Published online by Cambridge University Press

Lists and the `Maybe` type are instances of `MonadPlus`, thus:

```
instance  MonadPlus Maybe  where
    mzero                 = Nothing
    Nothing `mplus` ys    = ys
    xs      `mplus` ys    = xs

instance  MonadPlus []  where
    mzero = []
    mplus = (++)
```

20.3 Functions

The `join` function is the conventional monad join operator. It is used to remove one level of monadic structure, projecting its bound argument into the outer level.

The `mapAndUnzipM` function maps its first argument over a list, returning the result as a pair of lists. This function is mainly used with complicated data structures or a state-transforming monad.

The `zipWithM` function generalises `zipWith` to arbitrary monads. For instance the following function displays a file, prefixing each line with its line number,

```
listFile :: String -> IO ()
listFile nm =
  do cts <- readFile nm
     zipWithM_ (\i line -> do putStr (show i); putStr ": "; putStrLn line)
               [1..]
               (lines cts)
```

The `foldM` function is analogous to `foldl`, except that its result is encapsulated in a monad. Note that `foldM` works from left-to-right over the list arguments. This could be an issue where `(>>)` and the "folded function" are not commutative.

```
     foldM f a1 [x1, x2, ..., xm ]
==
     do
       a2 <- f a1 x1
       a3 <- f a2 x2
       ...
       f am xm
```

If right-to-left evaluation is required, the input list should be reversed.

The `when` and `unless` functions provide conditional execution of monadic expressions. For example,

https://doi.org/10.1017/S0956796803002211 Published online by Cambridge University Press

```
when debug (putStr "Debugging\n")
```

will output the string `"Debugging\n"` if the Boolean value `debug` is `True`, and otherwise do nothing.

The monadic lifting operators promote a function to a monad. The function arguments are scanned left to right. For example,

```
liftM2 (+) [0,1] [0,2] = [0,2,1,3]
liftM2 (+) (Just 1) Nothing = Nothing
```

In many situations, the `liftM` operations can be replaced by uses of `ap`, which promotes function application.

```
return f `ap` x1 `ap` ... `ap` xn
```

is equivalent to

```
liftMn f x1 x2 ... xn
```

20.4 Library `Monad`

```
module Monad (
    MonadPlus(mzero, mplus),
    join, guard, when, unless, ap,
    msum,
    filterM, mapAndUnzipM, zipWithM, zipWithM_, foldM,
    liftM, liftM2, liftM3, liftM4, liftM5,
    -- ...and what the Prelude exports
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    ) where

-- The MonadPlus class definition
class  (Monad m) => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a -> m a -> m a

-- Instances of MonadPlus
instance  MonadPlus Maybe  where
    mzero                = Nothing
    Nothing `mplus` ys   =  ys
    xs      `mplus` ys   =  xs

instance  MonadPlus []  where
    mzero =  []
    mplus = (++)
```

https://doi.org/10.1017/S0956796803002211 Published online by Cambridge University Press

```
-- Functions

msum             :: MonadPlus m => [m a] -> m a
msum xs          =  foldr mplus mzero xs

join             :: (Monad m) => m (m a) -> m a
join x           =  x >>= id

when             :: (Monad m) => Bool -> m () -> m ()
when p s         =  if p then s else return ()

unless           :: (Monad m) => Bool -> m () -> m ()
unless p s       =  when (not p) s

ap               :: (Monad m) => m (a -> b) -> m a -> m b
ap               =  liftM2 ($)

guard            :: MonadPlus m => Bool -> m ()
guard p          =  if p then return () else mzero

mapAndUnzipM     :: (Monad m) => (a -> m (b,c)) -> [a] -> m ([b], [c])
mapAndUnzipM f xs = sequence (map f xs) >>= return . unzip

zipWithM         :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys =  sequence (zipWith f xs ys)

zipWithM_         :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m ()
zipWithM_ f xs ys =  sequence_ (zipWith f xs ys)

foldM            :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
foldM f a []     =  return a
foldM f a (x:xs) =  f a x >>= \ y -> foldM f y xs

filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do { b  <- p x;
                        ys <- filterM p xs;
                        return (if b then (x:ys) else ys)
                      }

liftM            :: (Monad m) => (a -> b) -> (m a -> m b)
liftM f          =  \a -> do { a' <- a; return (f a') }

liftM2           :: (Monad m) => (a -> b -> c) -> (m a -> m b -> m c)
liftM2 f         =  \a b -> do { a' <- a; b' <- b; return (f a' b') }

liftM3           :: (Monad m) => (a -> b -> c -> d) ->
                                 (m a -> m b -> m c -> m d)
liftM3 f         =  \a b c -> do { a' <- a; b' <- b; c' <- c;
                                   return (f a' b' c') }

liftM4           :: (Monad m) => (a -> b -> c -> d -> e) ->
                                 (m a -> m b -> m c -> m d -> m e)
liftM4 f         =  \a b c d -> do { a' <- a; b' <- b; c' <- c; d' <- d;
                                     return (f a' b' c' d') }

liftM5           :: (Monad m) => (a -> b -> c -> d -> e -> f) ->
                                 (m a -> m b -> m c -> m d -> m e -> m f)
liftM5 f         =  \a b c d e -> do { a' <- a; b' <- b; c' <- c; d' <- d;
                                       e' <- e; return (f a' b' c' d' e') }
```

https://doi.org/10.1017/S0956796803002211 Published online by Cambridge University Press