The type constructor class Monad

```haskell
-- Imports necessary to compile this file in ghc
import Prelude hiding (Monad(..), Applicative(..), sequence, sequenceA,
                       sequence_, mapM, mapM_, guard, (<$>))
```

In the last chapter we learned how constructor classes can be defined and used. Now the question is: are there other useful type constructor classes? Therefore lets look at the `IO` type again. There are the following laws, which should be valid for `IO`.

```haskell
return () >> m = m
m >> return () = m
m >> (n >> o)  = (m >> n) >> o
```

They express that `(>>)` is associative and `return ()` is a neutral element. Hence `IO` with these two operations is a Monoid.

Corresponding laws hold for `return` and `(>>=)`:

```haskell
return v >>= \x -> m = m[x/v]
m >>= \x -> return x = m
m >>= \x -> (n >>= \y -> o) = (m >>= \x -> n) >>= \y -> o
```

For the last rule, we have to restrict, that the variable `x` may not occur in `o` as a free variable. [1].

Such a structure is called a Monad. The expression Monad comes from category theory, where they speak more generally of a functor and two natural transformations, which have to fulfil the monad laws[2]. The word Monad comes from Leibniz[3].

In Haskell a Monad is a unary type constructor `m` with the operations `return`, `(>>=)` (and `fail`), which fulfil the properties from above. They are represented by the class `Monad` defined as follows.

```haskell
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b

  return :: a -> m a

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

[1] The occurrence of a variable is free, if it was not introduced by a lambda. We will define this formally in the context of the lambda calculus.

[2] http://en.wikipedia.org/wiki/Monad_(category_theory)

[3] The word monad comes from Greek an means a single one. It was used in Leibniz' philosophical idea of Monadology, which tries to explain how the hole universe (everything) can be explained on basic, everlasting, impartial, unique spiritual atoms, called monads. See: https://en.wikipedia.org/wiki/Monadology

```haskell
(>>) :: m a -> m b -> m b
p >> q = p >>= \_ -> q
```

The do-Notation is syntactic sugar for all monads, not just for IO. So the next thing on our agenda is to get to know other monads.

The Maybe-Monad

Maybe is also a unary type constructor and (as for IO), there is a Functor instance for Maybe. Can we also define an instance for the class Monad?

```haskell
data Maybe a = Nothing | Just a
```

As an example we consider arithmetic expressions and want to evaluate them. Arithmetic expressions can be represented by the following algebraic data type.

```haskell
data Expr = Expr :+: Expr
          | Expr :/: Expr
          | Num Int
```

Unfortunately, the evaluation of an arbitrary expression may yield a run time error, if we have to divide by zero. Hence, evaluating such an expression should yield a Maybe value, as follows.

```haskell
eval (Num 3 :+: Num 4 ) -> Just 7
eval (Num 3 :/: (Num (-1) :+: Num 1)) -> Nothing
```

With this idea in mind, we define the function eval.

```haskell
eval :: Expr -> Maybe Int
eval (Num n) = Just n
eval (e1 :+: e2) = case eval e1 of
                     Nothing -> Nothing
                     Just n1 -> case eval e2 of
                                  Nothing -> Nothing
                                  Just n2 -> Just (n1 + n2)
eval (e1 :/: e2) = case eval e2 of
                     Nothing -> Nothing
                     Just 0  -> Nothing
                     Just n2 -> case eval e1 of
                                  Nothing -> Nothing
                                  Just n1 -> Just (n1 `div` n2)
```

The implementation becomes much simpler using Maybe as a Monad. The idea is, that the fault case Nothing stops the computation and yields Nothing independent of any other computation.

```haskell
instance Monad Maybe where
  Nothing >>= k = Nothing
  Just x  >>= k = k x

  return = Just

  fail _ = Nothing
```

Using the instance, it is much easier to define the eval function from above.

```haskell
eval :: Expr -> Maybe Int
eval (Num n) = return n
eval (e1 :+: e2) = do
  n1 <- eval e1
  n2 <- eval e2
  return (n1 + n2)
eval (e1 :/: e2) = do
  n2 <- eval e2
  if n2==0
    then Nothing
    else do
      n1 <- eval e1
      return (n1 `div` n2)
```

or without `do`-Notation:

```haskell
eval (Num n) = return n
eval (e1 :+: e2) =
  eval e1 >>= \n1 ->
  eval e2 >>= \n2 ->
  return (n1 + n2)
eval (e1 :/: e2) =
  eval e2 >>= \n2 ->
  if n2==0
    then Nothing
    else eval e1 >>= \n1 ->
         return (n1 `div` n2)
```

To prove the monad laws for `Maybe` is a simple exercise.

Before ghc 8, the class Monad was defined, as we introduced it in the last chapter. However, there exists a connection between the classes `Functor`, `Monad` and a class `Applicative`, which lays in some sense between these two classes. To understand this, let us first look at the connection between `Functor` and `Monad`. Is it possible, to defined the function `fmap` by means of the monad functions? At least for `Maybe`, we can define `fmap` as follows.

```haskell
fmap f ma = ma >>= \x -> return (f x)
```

or

```
fmap f ma = ma >>= return . f
```

Executing this code for other `Monad` instances, like `IO`, works as well. This definition is correct for arbitrary `Monad` instances and therefore, using this definition, the `Monad` laws imply the `Functor` laws.

`Functor` seems to be in some sense a smaller type class then `Monad`, which could be represented in the definition of the type constructor class Monad.

```
class Functor m => Monad m where
```

However, in ghc 8 the type hierarchy is a bit more complex and they introduced the class `Applicative` for Applicative Functors in between.

To understand the idea of `Applicative`, let us again consider the evaluation for the type `Expr`. A task in the eval function was the application of the function `(+)` to two values of type `Maybe`. Using `fmap`, we can only apply unary functions to values of type `Maybe`. Functions of arity two are not possible. Generalising the function `fmap` to a function of arity two would result in a function of the following type.

```
fmap2 :: Functor f => (a -> b -> c) -> f a -> f b -> f c
```

Unfortunately, this will not scale well, as we would get the following function for a function of arity three.

```
fmap3 :: Functor f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

Currying does not help us, representing the general case in one function. A solution is to pack or lift the function into the `Functor` class as well. This results into the operator `(<*>)`.

```
(<*>) :: Functor f => f (a -> b) -> f a -> f b
```

Using this operator, it is then possible, to add two `Maybe` values

```
Just (+) <*> ma <*> mb
```

where `ma` and `mb` are two arbitrary `Maybe` values. Note, that <*> binds left-associative. The function in the `Functor` class is successively applied to values in the Functor class. Curried functions of higher arity can successively be applied to there arguments within the same functor class.

Using this, we can redefine the `eval`-Function for the `:+:` case as follows.

```
eval (e1 :+: e2) = Just (+) <*> eval e1 <*> eval e2
```

To be more general, we can also generalise the function `Just` to a function like `return`, which in the context of applicative functors is called `pure`.

```
eval (e1 :+: e2) = pure (+) <*> eval e1 <*> eval e2
```

Combining all this, we obtain the class `Applicative`

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

and the corresponding `Maybe` instance

```
instance Applicative Maybe where
  pure                  = Just
  (Just f) <*> (Just x) = Just (f x)
  _        <*> _        = Nothing
```

as well as the following laws that have to hold for applicative functors.

```
pure id <*> v = v                              -- Identity
pure f <*> pure x = pure (f x)                 -- Homomorphism
u <*> pure y = pure ($ y) <*> u                -- Interchange
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

Proving these laws for `Maybe` is simple, and can be left as an exercise.

Similar to defining `fmap` by means of the monad functions, it is again possible to define `fmap` by means of the applicative functor

```
fmap f ma = pure f <*> ma
```

and again the `Functor` laws follow from the `Applicative` laws. Furthermore, there exists a synonym for `fmap`, called `(<$>)` which can be useful as an infix operator in the content of applicative functors, as we see in the definition of `eval` for the case `(:+:)`.

```
(<$>) :: Applicative f => (a -> b) -> f a -> f b
f <$> fx = pure f <*> fx
```

```
eval (e1 :+: e2) = (+) <$> eval e1 <*> eval e2
```

To show, that `Applicative` lays between `Functor` and `Monad`, we should now give definitions for the applicative function by means of the monadic functions. But again that is simple.

```
pure = return
mf (<*>) ma = mf >>= \f ->
              ma >>= \a ->
              return (f a)
```

For arbitrary monads, this function is defined in module `Control.Monad` as function `ap`. It can be used to avoid redundant definitions for the definitions of instances for `Functor`, `Applicative` and `Monad` within the same module.

However, it is not possible, to define the function `eval` in applicative style for the case `(:/:)`. The whole result depends on the result of the computation of `eval e2`. Hence, from an imperative point of view, the control flow depends on the result of `eval e2`. Applicative functors are not expressive enough to handle such situations, where the control flow depends on a pure value, i.e., the result of a computation.

Hence, the more expressive `Monad` has to be used.

```
class Applicative m => Monad m where
   (>>=) :: m a -> (a -> m b) -> m b

   return :: a -> m a

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

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

There are many useful functions defined for monads like `sequence` and `sequence_`.

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])
  where mcons p q = p >>= \x ->
                    q >>= \ys ->
                    return (x:ys)

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

```
    sequence [getLine, getLine] >>= print
    sequence [[1,2],[3,4]]
    ~> [[1,3],[1,4],[2,3],[2,4]]
```

And there is a variante working on `Applicative`.

```
sequenceA :: Applicative f => [f a] -> f [a]
sequenceA = foldr appcons (pure [])
  where appcons p q = (:) <$> p <*> q
```

It is also possible, to lift `map` to `Monads`.

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

```
    mapM_ putStr ["Hallo ", "Leute"]
    ~> Hallo Leute
```

```
mapM (\str -> putStr (str ++ ": ") >>
              getLine)
     ["Vorname", "Name"] >>= print
~> Vorname: $Frank$
   Name: $Huch$
   ["Frank","Huch"]
```

Again it is also possible to define this for `Applicative`. This function is called `traverse` and in ghc 8, it is combined with `sequenceA` and some other functions in the class `Traversable` (for more details see the library documentation).

```
traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
traverse f as = sequence (map f as)
```

Furthermore it is possible to define an `if-then` without `else` for `Monads`.

```
when :: Monad m => Bool -> m () -> m ()
when b a = if b then a else return ()

instance Monad IO where
    (>>=) = (>>=)
    return = return

main = do
  str <- getLine
  when (str == "Frank")
       (putStrLn "Hi Frank, nice to meet you")
  -- ...
```

Monad with a Plus

Another view on the data type `Maybe` is, that `Maybe` is a container, which can at most take one value. From this perspective, lists are a generalisation of `Maybe`, since they are containers of arbitrary capacity. It is also possible to define a `Monad` instance for lists:

```
instance Monad [] where
  return x = [x]
  -- return = (:[])

  (x:xs) >>= f = f x ++ (xs >>= f)
  []     >>= f = []
  -- (>>=) = flip concatMap

  fail _       = []
```

Then we get

```
[1,2,3] >>= \x -> [4,5] >>= \y -> return (x, y)
~> [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

and translated in `do`-notation.

```
do x <- [1,2,3]
   y <- [4,5]
   return (x, y)
```

The representation reminds strongly to list comprehensions, and in fact they are only syntactic sugar for the list monad.

```
[ (x, y) | x <- [1,2,3], y <- [4,5] ]
```

Another *property* of the list monad is, that the empty list is the zero of the structure.

```
m  >>= \_ -> [] = []
[] >>= \_ -> m  = []
```

Furthermore there is one distinguished, associative function `(++)` that holds the following law.

```
(m ++ n) ++ o = m ++ (n ++ o)
```

These two function can be combined in the class `MonadPlus`:

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

For lists we get the instance:

```
instance MonadPlus [] where
  mzero = []

  []     `mplus` ys = ys
  (x:xs) `mplus` ys = x : (xs `mplus` ys)
  -- mplus = (++)
```

We can also define an instance for `Maybe`:

```
instance MonadPlus Maybe where
  mzero = Nothing

  Nothing  `mplus` m = m
  (Just x) `mplus` _ = Just x
```

Searching with the List Monad

As we saw before, the monad list and list comprehensions are similar structures. So far, we do not know, how to express boolean conditions in the list monad, as we do using list comprehensions.

```
[ (x, y) | x <- [1,2,3], y <- [2,3,4], x==y ]
```

To implement such a condition, we can use the function `guard`.

```
guard :: Bool -> [()]
guard False = []     -- zero
guard True  = [()]   -- one
```

Thinking in backtracking or a search algorithm, the function `guard` restricts the search and cuts a possible branch.

To understand, how `guard` works, let us consider the following example

```
return False >>= guard >> return 42
```

which is reduced as

```
  guard False >> return 42
= [] >> return 42
= []
```

in contrast the expression

```
return True >>= guard >> return 42
```

reduces as follows.

```
  guard True >> return 42
= [()] >> return 42
= return 42
= [42]
```

Hence, with `guard` it is possible to refuse a result in dependence of a (sub-)result. This reminds us of the `filter` function, which can easily be defined using `guard`.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = do x <- xs
                 guard (p x)
                 return x
```

Usually, `guard` is used to select valid results from a larger search space. For instance, a program for computing Pythagorean triples can be defined as follows.

```
pytriples :: Int -> [(Int,Int,Int)]
pytriples max = do a <- [1..max]
```

```
                  b <- [a..max]
                  c <- [b..max]
                  guard (a*a + b*b == c*c)
                  return (a,b,c)
```

The call `pytriples 10` yields `[(3,4,5),(6,8,10)]`.

In fact, it is possible to translate arbitrary list comprehensions into `do`-notation (exercise).

The `guard` function is defined for arbitrary instances of class `MonadPlus`.

```
guard :: MonadPlus m => Bool -> m ()
guard False = mzero
guard True  = return ()
```

With this generalised type, we also get a generalised type for the Pythagorean triples

```
pytriples :: MonadPlus m => Int -> m (Int,Int,Int)
```

and can execute the function in any `MonadPlus` instance.

Laws for `MonadPlus`

So far, we defined the laws for `MonadPlus` only for lists. In general they are:

```
m     >>= \_ -> mzero = mzero
mzero >>= \_ -> m     = mzero

(m `mplus` n) `mplus` o = m `mplus` (n `mplus` o)
```

`mzero` und `mplus` built a monoid. However, there is another property you expect for `MonadPlus` instances. For every function `f` the usage of `(>>= f)` should distribute with respect to `mzero` und `mplus` (so in some sense be a `MonadPlus` homomorphism):

```
        mzero >>= f   =   mzero
(a `mplus` b) >>= f   =   (a >>= f) `mplus` (b >>= f)
```

This distributive law, guarantees, that during the computation no results get lost. Unfortunately, they are not fulfilled for every predefined `MonadPlus` instance. For example, the implementation for `Maybe` does not fulfill the distributivity law for `MonadPlus`. For `a = return False`, `b = return True` and `f = guard`, we get:

```
  (return False `mplus` return True) >>= guard
= (Just False `mplus` Just True) >>= guard
= Just False >>= guard
= guard False
= Nothing
```

but

```
   (return False >>= guard) `mplus` (return True >>= guard)
 = guard False `mplus` guard True
 = Nothing `mplus` Just ()
 = Just ()
```

As a consequence, using the `Maybe` monad to solve search problems does not always work. For instance, executing the Pythagorean triples in the `Maybe` monad, we obtain the result `Nothing` instead of `Just (3,4,5)`.