Paradigmas de Programação

Week 9
Monads
(Continued...)[1]

Alexandra Mendes

[1]These slides use the information presented in https://wiki.haskell.org/All_About_Monads.

What is a Monad?

- A specific way of chaining operations together.
- A strategy for combining computations into more complex computations.
- A way to structure computations in terms of values and sequences of computations using those values.
- Monads allow the programmer to build up computations using sequential building blocks, without having to code the combination each time it's needed.

What is a Monad?

- Monads are used to get imperative behaviour (IO, State, Error, etc) out of functional programs.
- Allows computations to be isolated from side effects and non-determinism, thus remaining purely functional.

What is a Monad?

- Monads are used to get imperative behaviour (IO, State, Error, etc) out of functional programs.
- Allows computations to be isolated from side effects and non-determinism, thus remaining purely functional.
- Remember: in Haskell, functions are **pure**, i.e.:
    - A function returns exactly the same result every time it's called with the same set of arguments.
    - A function has no state, nor can it access any external state.
    - A function has no side effects. Calling a function once is the same as calling it twice and discarding the result of the first call.

What is a Monad?

- Monads are used to get imperative behaviour (IO, State, Error, etc) out of functional programs.
- Allows computations to be isolated from side effects and non-determinism, thus remaining purely functional.
- Remember: in Haskell, functions are **pure**, i.e.:
  - A function returns exactly the same result every time it's called with the same set of arguments.
  - A function has no state, nor can it access any external state.
  - A function has no side effects. Calling a function once is the same as calling it twice and discarding the result of the first call.
- Take as an example the function `getChar:: IO char`. Can we be sure how it will behave every time we call it?

What is a Monad?

- Monads are used to get imperative behaviour (IO, State, Error, etc) out of functional programs.
- Allows computations to be isolated from side effects and non-determinism, thus remaining purely functional.
- Remember: in Haskell, functions are **pure**, i.e.:
  - A function returns exactly the same result every time it's called with the same set of arguments.
  - A function has no state, nor can it access any external state.
  - A function has no side effects. Calling a function once is the same as calling it twice and discarding the result of the first call.
- Take as an example the function `getChar::  IO char`. Can we be sure how it will behave every time we call it?
  - No. The result of calling getChar depends on the input typed by the user.

Meet the Monads

Type Constructors:

A parameterized type definition used with polymorphic types, e.g:

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

`Maybe` is a type constructor and `Nothing` and `Just` are data constructors.

Construct a **data value**:

```
country = Just "Portugal"
```

Construct a **type** by applying the Maybe type constructor to a type, e.g. `Maybe Int`.

Meet the Monads

Type Constructors

- Polymorphic types can be seen as containers that are capable of holding values of many different types: `Maybe Int` can be thought of as a `Maybe` container holding an `Int` value (or Nothing)
- In Haskell, we can also make the **type of the container polymorphic**: `m a` represents a container of some type holding a value of some type!

Meet the Monads

In Haskell a monad is represented as:

- A type constructor (call it m)
- A function that builds values of that type $(a \to m\ a)$
- And a function that combines values of that type with computations that produce values of that type to produce a new computation for values of that type $(m\ a \to (a \to m\ b) \to m\ b)$.

The container is the same but the type of its contents can change.

Meet the Monads

In Haskell a monad is represented as:

- A type constructor (call it m)
- A function that builds values of that type
  $(a \to m\ a) \leftarrow$ called `return`
- And a function that combines values of that type with computations that produce values of that type to produce a new computation for values of that type
  $(m\ a \to (a \to m\ b) \to m\ b)$. $\leftarrow$ called `bind` (`>>=`)

The container is the same but the type of its contents can change.

Meet the Monads

```
-- the type of monad m
data m a = ...

-- return takes a value and embeds it in the monad.
return :: a -> m a

-- bind is a function that combines a monad instance m a
-- with a computation that produces another monad
-- instance m b from a's to produce a new monad
-- instance m b
(>>=) :: m a -> (a -> m b) -> m b
```

Meet the Monads

Put simply:

- The monad type constructor defines a type of computation.
- The return function creates primitive values of that computation type.
- >>= combines computations of that type together to make more complex computations of that type.

Meet the Monads

Using the container analogy:

- The type constructor `m` is a container that can hold different values.
- `m` a is a container holding a value of type a.
- The return function puts a value into a monad container.
- The >>= function takes the value from a monad container and passes it to a function to produce a monad container containing a new value, possibly of a different type.
- The >>= function is known as "bind" because it binds the value in a monad container to the first argument of a function.

Bind: $>>=$

The following:

```
m a >>= \v -> m b
```

combines a monadic value `m a` containing values of type `a` and a function which operates on a value `v` of type `a`, returning the monadic value `m b`.

Bind: >>=

The following:

```
m a >>= \v -> m b
```

combines a monadic value `m a` containing values of type `a` and a function which operates on a value `v` of type `a`, returning the monadic value `m b`.

Exercise

How can you combine the `getLine` with the `putStrLn` so that the line read by `getLine` is printed by `putStrLn`?

Try the following in the command line: `getLine >>= putStrLn`

An Example

Suppose that we are writing a program to keep track of sheep cloning experiments. We want to keep the genetic history of all of our sheep, so we would need mother and father functions. But since these are cloned sheep, they may not always have both a mother and a father!

We would represent the possibility of not having a mother or father using the Maybe type constructor in our Haskell code:

```
type Sheep = ...

father :: Sheep -> Maybe Sheep
father = ...

mother :: Sheep -> Maybe Sheep
mother = ...
```

An Example

Then, defining functions to find grandparents is a little more complicated, because we have to handle the possibility of not having a parent:

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = case (mother s) of
                          Nothing -> Nothing
                          Just m  -> father m
```

and so on for the other grandparent combinations.

An Example

It gets even worse if we want to find great grandparents:

```
mPGrandfather :: Sheep -> Maybe Sheep
mPGrandfather s = case (mother s) of
                     Nothing -> Nothing
                     Just m  -> case (father m) of
                                   Nothing -> Nothing
                                   Just gf -> father gf
```

- It is clear that a `Nothing` value at any point in the computation will cause Nothing to be the final result;
- It would be much nicer to implement this notion once in a single place and remove all of the explicit case testing scattered all over the code.

An Example

```haskell
-- comb is a combinator for sequencing
-- operations that return Maybe

comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing  _ = Nothing
comb (Just x) f = f x

-- now we can use `comb` to build
-- complicated sequences

mPGrandfather :: Sheep -> Maybe Sheep
mPGrandfather s = (Just s) `comb` mother `comb`
                                  father `comb` father
```

Code is much cleaner and easier to write, understand and modify.

The comb function is polymorphic, not specialized for Sheep.

An Example

We have now created a monad!

The Maybe type constructor along with the Just function (acts like return) and our combinator (acts like >>=) together form a simple monad for building computations which may not return a value.

All that remains to make this monad truly useful is to make it conform to the monad framework built into the Haskell language.

Standard Monad Class Definition

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

Standard Monad Class Definition

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

- It is not strictly necessary to make your monads instances of the class but it is a good idea.

Standard Monad Class Definition

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

- It is not strictly necessary to make your monads instances of the class but it is a good idea.
- Not only `Haskell` has special support for Monad instances built into the language but making your monads instances of the Monad class communicates important information to others who read the code.

Back to Maybe

How does the Maybe type constructor fit into the Haskell monad framework?

```
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

Back to Maybe

How does the Maybe type constructor fit into the Haskell monad framework?

```
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

Without using GHCi, what is the output of each of the following

```
Just 6 >>= (\ x -> if (x == 0) then fail "zero"
                               else Just (x * 2))
Just 0 >>= (\ x -> if (x == 0) then fail "zero"
                               else Just (x + 3) )
Nothing >>= (\ x -> if (x == 0) then fail "zero"
                                else Just (x * 4) )
```

Back to our example:

```
-- we can use monadic operations
-- to build complicated sequences

maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s =
                   (return s) >>= mother >>= father

fathersMaternalGrandmother :: Sheep -> Maybe Sheep
fathersMaternalGrandmother s =
        (return s) >>= father >>= mother >>= mother
```

Maybe is an instance of Monad in the standard prelude.

Functions with Monads

A function of the type:

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

is much more flexible than one of the type

```
doSomething :: a -> Maybe b
```

When possible, make use of the Monad class instead of using a specific monad instance when writing a function.

Do-notation

- Another advantage of membership in the Monad class is the support for do-notation.
- The do-notation is an expressive shorthand for building up monadic computations, using a pseudo-imperative style with named variables.
- The result of a monadic computation can be “assigned” to a variable using a left arrow $\leftarrow$ operator.
- Then using that variable in a subsequent monadic computation automatically performs the binding.

Do-notation

- The type of the expression to the right of the arrow is a monadic type `m a`.
- The expression to the left of the arrow is a pattern to be matched against the value inside the monad.
- `(x:xs)` would match against `Maybe [1,2,3]`.

Do-notation

- The type of the expression to the right of the arrow is a monadic type `m a`.
- The expression to the left of the arrow is a pattern to be matched against the value inside the monad.
- `(x:xs)` would match against `Maybe [1,2,3]`.

Example

```
-- we can also use do-notation
-- to build complicated sequences

mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = do m  <- mother s
                                  gf <- father m
                                  father gf
```

Compare this to previous `fathersMaternalGrandmother`.

Do-notation

Version 1

```haskell
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s =
        (return s) >>= mother >>= father >>= father
```

Version 2

```haskell
-- we can also use do-notation
-- to build complicated sequences

mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = do m  <- mother s
                                  gf <- father m
                                  father gf
```

Do-notation

- The do-block shown before is written using the layout rule to define the extent of the block.
- Haskell also allows you to use braces and semicolons when defining a do-block:

```
mothersPaternalGrandfather s =
        do {m <- mother s; gf <- father m; father gf}
```

Do-notation

- The do-block shown before is written using the layout rule to define the extent of the block.
- `Haskell` also allows you to use braces and semicolons when defining a do-block:

```
mothersPaternalGrandfather s =
        do {m <- mother s; gf <- father m; father gf}
```

- Do-notation is simply syntactic sugar!
- There is nothing that can be done using do-notation that cannot be done using only the standard monadic operators.
- It can be more convenient when the monadic computation is long.

Do-notation to Monadic Operators

- Every expression matched to a pattern:

  ```
  x <- expr1
  ```

  becomes

  ```
  expr1 >>= \x ->
  ```

- Every expression without a variable assignment, `expr2` becomes

  ```
  expr2 >>= \_ ->
  ```

- All do-blocks must end with a monadic expression.

Do-notation to Monadic Operators

The definition of `mothersPaternalGrandfather` above would be translated to:

```
mothersPaternalGrandfather s = mother s >>= (\m ->
                               father m >>= (\gf ->
                               father gf))
```

(The parentheses are for clarity and aren't actually required.)

The binding operator is so named because it is literally used to bind the value in the monad to the argument in the following lambda expression.

The Monad Laws

Monadic operations must obey a set of laws.

The Monad Axioms

To be a proper monad, the return and >>= functions must work together according to three laws:

```
1. (return x) >>= f == f x
2. m >>= return == m
3. (m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

Law 3 can be re-written, for clarity, as:

```
(m >>= (\x -> f x)) >>= g == m >>= (\x -> f x >>= g)
```

The Monad Laws

- The first law requires that return is a left-identity with respect to >>=.
- The second law requires that return is a right-identity with respect to >>=.
- The third law is a kind of associativity law for >>=.
- Obeying the three laws ensures that the semantics of the do-notation using the monad will be consistent.

Any type constructor with return and bind operators that satisfy the three monad laws is a monad.

What is the practical meaning of the monad laws?

Left identity:

```
(return x) >>= f == f x
```

```
do { x' <- return x;
     f x'
   }
```

equivalente a:

```
do { f x }
```

What is the practical meaning of the monad laws?

Right identity:

```
m >>= return == m
```

```
do { x <- m;
     return x
   }
```

equivalente a:

```
do { m }
```

What is the practical meaning of the monad laws?

Associativity:

```
(m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

```
do { y <- do { x <- m;
               f x
             }
     g y
   }
```

equivalente a:

```
do { x <- m;
     do { y <- f x;
          g y
        }
   }
```

que por sua vez é equivalente a:

```
do { x <- m;
     y <- f x;
     g y
   }
```

But why should monads obey these laws?

The laws appear as plain common-sense transformations of imperative programs.

In practice, people do write like the lengthier left-hand side once in a while.

Example: Beginners tend to write

```
skip_and_get = do
                  unused <- getLine
                  line <- getLine
                  return line
```

and it should behave in the same way as

```
skip_and_get = do
                  unused <- getLine
                  getLine
```

But why should monads obey these laws?

Another example: when you use *skip_and_get*

```
main = do
         answer <- skip_and_get
         putStrLn answer
```

which is the same as:

```
main = do
         answer <- do
                     unused <- getLine
                     getLine
         putStrLn answer
```

and applying associativity:

```
main = do
         unused <- getLine
         answer <- getLine
         putStrLn answer
```

Failure IS an option

The full definition of the Monad class actually includes two additional functions: fail and $>>$.

The default implementation of the fail function is:

```
fail s = error s
```

You do not need to change this for your monad unless you want to provide different behavior for failure or to incorporate failure into the computational strategy of your monad.

For example, in the Maybe monad:

```
fail _ = Nothing
```

Failure IS an option

The fail function is not a required part of the mathematical definition of a monad; It is included in the standard Monad class definition because of the role it plays in Haskell's do-notation.

The fail function is called whenever a pattern matching failure occurs in a do-block:

```
fn :: Int -> Maybe [Int]
fn idx = do
        let l = [Just [1,2,3], Nothing, Just [7..20]]
        (x:xs) <- l!!idx -- a pattern match failure will
        return xs
```

fn 0 has the value *Just* $[2, 3]$, but *fn* 1 has the value Nothing.

The Function >>

The function >> is a convenience operator that is used to bind a monadic computation that does not require input from the previous computation in the sequence.

It is defined in terms of >>=:

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

Zero and Plus

Some monads obey additional laws. These monads have a special value `mzero` and an operator `mplus` that obey four additional laws:

```
1. mzero >>= f == mzero
2. m >>= (\x -> mzero) == mzero
3. mzero `mplus` m == m
4. m `mplus` mzero == m
```

It is easy to remember the laws for `mzero` and `mplus` if you associate `mzero` with 0, `mplus` with $+$, and $>>=$ with $\times$ in ordinary arithmetic.

Zero and Plus

Monads which have a zero and a plus can be declared as instances of the `MonadPlus` class in Haskell:

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

Zero and Plus

For example, the Maybe monad:

```
instance MonadPlus Maybe where
    mzero                = Nothing
    Nothing `mplus` x = x
    x `mplus` _          = x
```

- Nothing is the zero value and adding two Maybe values together gives the first value that is not Nothing.
- If both input values are Nothing, then the result of mplus is also Nothing.

mplus Operator

The mplus operator is used to combine monadic values from separate computations into a single monadic value.
In our sheep-cloning example:

```
parent s = (mother s) `mplus` (father s)
```

would return a parent if there is one, and Nothing is the sheep has no parents at all.

For a sheep with both parents, the function would return one or the other, depending on the exact definition of mplus in the Maybe monad.

List is also a Monad

- The List monad allows us to build computations which can return 0, 1, or more values.
- The List monad thus embodies a strategy for performing simultaneous computations along all allowed paths of an ambiguous computation.
- The return function creates a singleton list (return x = [x]).
- The binding operation for creates a new list containing the results of applying the function to all of the values in the original list: `(l >>= f = concatMap f l)`.
- The List monad also has a zero and a plus. `mzero` is the empty list and `mplus` is the ++ operator.

List Monad

```
instance Monad [] where
    m >>= f   = concatMap f m
    return x = [x]
    fail s   = []

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