Lecture 10: Monads

Applications to programming and categorical context

Luis Barbosa and Renato Neves

2 de Dezembro de 2019

1 Summary

The notion of monad has been extensively studied in universal algebra since the sixties [Lin66]. In the late eighties, when E. Moggi proposed to use it as a *uniform semantics* for programming languages [Mog89, Mog91], it began to be understood also as an important concept of computer science – the idea was later introduced into the programming practice by P. Wadler, which lead to a rigorous style of combining purely functional programs that can mimic side-effects [Wad95]. The core observation in [Mog89, Mog91] is that monads encode in abstract terms several kinds of *computational effect*. This includes exceptions, state updates, timing constraints, nondeterminism, and probabilistic behaviour. Under this view, a computational effect is given by a type constructor T (technically, an endofunctor) and computations are elements of TY for some type Y. For example, the datatype "Maybe" in Haskell, which is defined by,

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

corresponds to one such constructor T. A program p is then regarded as a function $[\![p]\!] : X \longrightarrow TY$ producing a computational effect for each input in X, and sequential composition of programs is *automatically* handled by the monadic structure associated to the computational effect (see more details below) .

In this lecture we go over the the following topics:

(1) Motivation for the use of monads in program semantics: the Hoover "damn" and the Knight's quest.

(2) Definition of a monad (as a Kleisli triple) and examples.

(3) Illustrations in the programming language Haskell.

(4) The do-notation and the notion of commutative monad.

(5) Equivalent definitions of a monad.

(6) A very brief introduction to monad combination.

(7) The connection between adjunctions and monads.

Bibliography: A great first introduction to monads is available in the book *Learn You a Haskell for Great Good* by Miran Lipovača. You can download it here. The article *Monads for functional programming* by Philip Wadler is another great source of intuitions, theory and applications of monads. You can

download it here. Finally, a more categorical study of a monad (and related results) is available in the book *Category Theory* by Steve Awodey.

Assessment: The following sections contain a series of exercises for you to solve. We expect to receive until the *end of December* the solutions to these exercises by email and in a single PDF file properly identified with your name and student number. Please do *not forget* to thoroughly explain how each proposed solution was reached. If you get stuck in one of the exercises try to move foward and then come back to it later on.

2 The Hoover "damn"

The Hoover dam is a famous, very big dam in the U.S. that regulates the water flow of the Colorado river in the Black Canyon. Until a few decades ago, in order to get from Las Vegas to Arizona by car we would need to cross the Hoover dam. However, the dam is old and the engineer responsible for it told us that only *three cars* can be on top of it at the same time. In the class, we defined the following functions in Haskell to count the number of cars on top of the dam at the same time.

```
-- A car reaches the top of the dam. There can only be three cars
-- on the dam at the same time
carEnters :: Int -> Maybe Int
carEnters x = if (x < 3) then Just (x+1) else Nothing

-- A car leaves the top of the dam.
carLeaves :: Int -> Maybe Int
carLeaves x = if (x > 1) then Just (x - 1) else Just 0

-- The dam opens (initial state)
damOpens :: Maybe Int
damOpens = Just 0
```

Recall that the value `Nothing` was used to inform that the dam collapsed. We then simulated what happens when 'two cars enters the dam' followed by 'two cars leave the dam', via the program,

```
damOpens >>= carEnters >>= carEnters >>= carLeaves >>= carLeaves
```

and similarly we simulated the scenario,

```
damOpens >>= carEnters >>= carEnters >>= carEnters >>= carEnters >>=
carLeaves >>= carLeaves
```

EXERCISE 1. Tell and justify what is the returning value of this last program.

Our simulation of cars entering and leaving the dam heavily relied on the *monadic bind* operation, which we saw in the class:

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

In a nutshell, given a *computation* `m a` of type `a` and a function `a -> m b` returning computations, the monadic bind operation returns a computation `m b` of type `b`. Internally, the monadic bind operation achieves this by "automagically" turning a function of type `a -> m b` into a function `m a -> m b` and then feeding the computation `m a` into the latter. This "magic" is precisely the essence of a monad.

Definition 1. *Given a category* C*, a monad is a triple* $(T, \eta, (-)^*)$ *where* $T : \mathsf{Obj}\mathsf{C} \longrightarrow \mathsf{Obj}\mathsf{C}$ *is a function between the objects of the category* C*,* η *– which is called the* unit *of the monad – is a morphism* $\eta : X \longrightarrow TX$ *for every object* X *in* C*, and* $(-)^*$ *– which is called* Kleisi lifting *of a monad (its magic) – sends a morphism of type* $f : X \longrightarrow TY$ *into a morphism of type* $f^* : TX \longrightarrow TY$*. Moreover, the following equations must hold for every morphism* $f : X \longrightarrow TY$ *and* $g : Y \longrightarrow TZ$*:*

$$\eta^*_X = \mathrm{id}_{TX}, \qquad f^* \cdot \eta_X = f, \qquad (g^* \cdot f)^* = g^* \cdot f^*$$

EXERCISE 2. Show that the functor $(- + 1)$ is a monad. What is its relationship with the datatype `Maybe`?

After awhile, the engineer responsible for the dam came to us and told that, after all, is *not certain* whether the dam will collapse if *four cars* are on top of it at the same time (for more than four cars it is still certain that the dam will collapse; and for less than four it is still certain that the dam will not collapse). We need therefore to fit this new piece of information into our simulation program. In particular, we will now need to use the datatype,

```
data MaybeList a = MaybeList [Maybe a]
```

in lieu of the datatype `Maybe`. Note that the datatype `MaybeList` involves lists which (as we saw in the lecture) can be used to encode *nondeterminism*. Next, we need to,

EXERCISE 3. Prove that this new datatype forms a monad (*hint*: start by proving that the datatype `List` forms a monad).

EXERCISE 4. Implement the respective monad in the language Haskell (by completing the code below)

```
data MaybeList a = MaybeList [Maybe a]

instance Functor MaybeList where
  fmap f x = ...

instance Applicative MaybeList where
  pure x = ...
  f <*> x = ...

instance Monad MaybeList where
    x >>= k = ...
    return x = ...
```

EXERCISE 5. Complete the functions below by taking into account the new information given by the engineer.

```
-- A car reaches the top of the dam. There can only be three cars
-- on the dam at the same time
carEnters :: Int -> MaybeList Int
carEnters = ...

-- A car leaves the top of the dam.
carLeaves :: Int -> MaybeList Int
carLeaves = ...

-- The dam opens (initial state)
damOpens :: MaybeList Int
damOpens = [Just 0]
```

And finally, what do the two computations below tell us? (No need to write down the answer).

```
damOpens >>= carEnters >>= carEnters >>= carLeaves >>= carLeaves

damOpens >>= carEnters >>= carEnters >>= carEnters >>= carEnters >>=
carLeaves >>= carLeaves
```

Warning: this section's remainder contains a much harder problem. You do not address it in order to obtain the maximum grade.

Scratching his head, the engineer came back to us with a much more refined information. He tells us the following: "the probability of the dam collapsing when a car enters is given by the formula,

$$p = \min(1, n/10)$$

where n is the number of cars currently on the dam". Do not ask him how he got this new information, because he will be upset. Instead ask yourself, can we tackle this new piece of information with a new monad? It turns out that indeed we can: using the monad of subdistributions. A slightly simpler version of this monad is thoroughly detailed here. Solve the last three exercises via this new monad and observe what you obtain for:

```
damOpens >>= carEnters >>= carEnters >>= carLeaves >>= carLeaves

damOpens >>= carEnters >>= carEnters >>= carEnters >>= carEnters >>=
carLeaves >>= carLeaves
```

3 The do-notation and the notion of commutative monad

As a reward for our splendid job on the Hoover dam, the engineer gave us a pocket calculator! However, the calculator has a strong personality and also a strong *vendetta* against number 3: among other things, it forces a division returning 3 to return `Nothing` instead. The code representing this thorny calculator is presented next.

```
-- A function for dividing numbers. Note the if-clause.
myDiv :: Integral a => a -> a -> Maybe a
myDiv a b = if (b /= 0 && (div a b) /= 3) then Just (div a b) else Nothing

-- A function for adding numbers.
mySum :: Integral a => a -> a -> Maybe a
mySum a b = Just (a + b)

-- A function for multiplying numbers
myMult :: Integral a => a -> a -> Maybe a
myMult a b = Just (a * b)
```

EXERCISE 6. Use these three functions and the respective monadic binding (>>=) to calculate the expression $(a/b) + (b/a)$ (fill-in the code below). Hint: use the notion of λ-abstraction, which you can consult here and in previous lecture notes.

```
calc (a,b) = ...
```

The do-notation is a common mechanism to render programs with multiple arguments more readable. Formally, the do-program below on the left unfolds into the the one below on the right.

```
do x <- p
   q(x)
```

```
p >>= (\x -> q(x))
```

EXERCISE 7. Use the correspondence above to convert your program `calc` into a do-program

```
calc (a,b) = do x <- ...
```

and prove that the program thus obtained is equivalent to the previous one.
A monad is called *commutative* when for every three programs p, q, r the following two programs are the same.

```
do x <- p
   y <- q
   r x y
```

```
do y <- q
   x <- p
   r x y
```

This is another useful mechanism for simplifying programs and turning them more readable. Unfortunately, not all monads are commutative. A prime example of this is the monad that we will study next.

Definition 2. *Consider a set S. The state monad is the triple $(((-) \times S)^S, \eta, (-)^*)$ such that $\eta_X(x) = \lambda s.(x, s)$ and the Kleisli lifting is defined by,*

$$f^*(\langle a, b\rangle) = \lambda s.\, f(a(s))\, b(s)$$

where $\langle a, b\rangle : S \longrightarrow X \times S$ is a function.

The Kleisli lifting of the state monad encodes the following sequence of events: 1) starting with an initial state s, we extract a *new state* $b(s) \in S$ and a *value* $a(s) \in X$. 2) We feed the value $a(s) \in X$ to f and thus obtain a new function $f(a(s)) : S \longrightarrow Y \times S$. 3) Then, in order to extract a value of type Y from this function, we feed it with the last state obtained, namely $b(s)$. This behaviour of the Kleisli lifting is tremendously useful for composing programs with internal memory, and is best understood with examples: so consider the following code.

```
import Control.Monad.State.Lazy -- we need to import this library in order to use the state
-- monad. Also, note that this library provides the following functions for retrieving
-- information from the memory s and for writing information to the memory s.
-- get :: State s s
-- put :: s -> State s ()
-- Before moving forward, try to write down the definition of such functions yourself.
incInternalCounter :: State Int ()
incInternalCounter = do n <- get
                        put (n+1)

doubIncInternalCounter :: State Int ()
doubIncInternalCounter = incInternalCounter >> incInternalCounter
```

EXERCISE 8. Tell and justify what is the purpose of the two programs above. If you are having difficulties, here are some hints to help you. Hint 1: see more details about the *binding operator*

```
>> :: m a -> m b -> m b
```

for example here. Hint 2: the symbol `()` corresponds to the singleton set 1. Hint 3: the datatype `State s a` corresponds to functions of the type `s -> (a,s)`. Hint 4: to test your conclusions, you will need to use the function,

```
runState :: State s a -> s -> (a, s)
```

For example the program,

```
runstate incInternalCounter 0
```

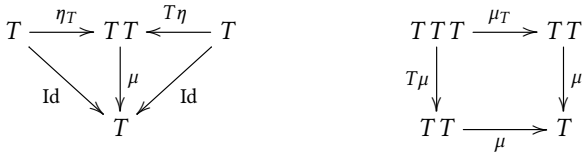
gives the output of `incInternalCounter` with `0` as the *initial state*.

EXERCISE 9. Find an example of program that shows that the state monad is *non-commutative*.

4 An equivalent definition of a monad

In the lectures we studied the definition of a monad in terms of a Kleisli triple and another definition of a monad in terms or functors and natural transformations. Let us recall the latter case.

Definition 3. *A monad on a category* C *is a triple* (T, η, μ) *where* $T : \mathsf{C} \longrightarrow \mathsf{C}$ *is a functor,* $\eta : \mathrm{Id} \longrightarrow T$ *is a natural transformation, and* $\mu : TT \longrightarrow T$ *is a natural transformation that make the following diagrams commute.*



EXERCISE 10. Prove that both definitions are equivalent.

The latter definition of a monad makes some structures more evident and eases certain constructions. Let us analyse some examples of this aspect. In the category Set (of sets and functions) every functor comes equipped with a natural transformation (called its *strength*),

$$\mathrm{str}_{X,Y} : TX \times Y \longrightarrow T(X \times Y)$$

defined by the mapping,

$$(t, y) \mapsto T(\lambda x \in X.(x, y))\, t$$

EXERCISE 11. Give an explicit definition of this natural transformation for the functors $(+1)$, $\mathbb{N}_0 \times (-)$ and $[-]$.

This natural transformation is very useful for combining monads. For example,

EXERCISE 12. Use this natural transformation to show that the functor $[String \times (-)]$ is also a monad. Hint: note that $String$ forms a monoid when equipped with the concatenation operation and the empty list.

In the lectures we studied the problem of the *Knight's quest*. Briefly, given a Knight in a chessboard (the latter seen as a grid) the problem consisted in knowing whether the knight can reach a target position within a certain number of steps (recall that a Knight can only move in certain ways). We considered the following code to represent this problem.

```
-- The Knight's Quest (more details in "Learn you a Haskell for Great Good")
possibleMoves :: (Int,Int) -> [(Int,Int)]
possibleMoves (x,y) = [(x-1,y+2),(x+1,y+2),(x-1,y-2),(x+1,y-2),
                       (x-2,y+1),(x-2,y-1),(x+2,y+1),(x+2,y-1)]

-- The starting position
initialMove :: [(Int,Int)]
initialMove = [(0,0)]

-- Determines whether the target position was achieved or not
targetAchieved :: (Int,Int) -> [(Int,Int)] -> Bool
targetAchieved = elem
```

Then we used the code,

```
targetAchieved (0,1) $ initialMove >>= possibleMoves >>= possibleMoves >>= possibleMoves
```

to determine whether the Knight can reach the target position in three or less steps. A quick observation of this code, however, tells us that the Knight is amnesic! He cannot remember what was his previous position. This is very sad, because even if he reaches his target position he cannot remember which path he used to get there. To fix this, we will give him a logbook. More technically, we will consider the monad,

```haskell
data LogList a = LogList [(String,a)]
```

and use the solution to Problem 12 in our favour.

EXERCISE 13. Implement the `LogList` monad in the language HASKELL (by completing the code below)

```haskell
instance Functor LogList where
  fmap f x = ...

instance Applicative LogList where
  pure x = ...
  f <*> x = ...

instance Monad LogList where
    x >>= k = ...
    return x = ...
```

EXERCISE 14. Complete the code below by taking into account that the Knight can now register his moves.

```haskell
-- The Knight's Quest
possibleMoves :: (Int,Int) -> LogList (Int,Int)
possibleMoves = ...

-- Determines whether the target position was achieved or not
targetAchieved :: (Int,Int) -> LogList (Int,Int) -> Bool
targetAchieved = ...

-- Determines whether the target position was achieved or not and the path taken
-- to reach this position
targetAchievedWithPath :: (Int,Int) -> LogList (Int,Int) -> Maybe String
targetAchievedWithPath = ...
```

Starting in $(0,0)$, what is the path that the Knight should take to reach the position (1,0) in three or less steps? (No need to write down the answer).

5 Monads and adjunctions

Warning: this section contains problems connecting monads and adjunctions. You do not address them in order to obtain the maximum grade.

Monads are closely related to adjunctions (recall the previous lecture on the latter). We will now analyse this connection in some detail. Let us start by noting that every monad $\mathbb{T} = (T, \eta, \mu)$ induces a category as described next.

Definition 4. *The Kleisli category* $\mathsf{C}_{\mathbb{T}}$ *of a monad* $\mathbb{T}$ *has as objects those of* C *and as hom-sets those defined by the equation,*

$$\mathsf{C}_{\mathbb{T}}(X, Y) = \mathsf{C}(X, TY)$$

For each $\mathsf{C}_{\mathbb{T}}$*-object* X *the identity is* $\eta_X : X \longrightarrow TX$*, and the composition* $g \bullet f$ *of two morphisms* $f : X \longrightarrow TY$ *and* $g : Y \longrightarrow TZ$ *is* $\mu_Z \cdot Tg \cdot f$*. This is often called the category of programs of the monad* $\mathbb{T}$*.*

There exists a functor $\mathsf{C} \longrightarrow \mathsf{C}_{\mathbb{T}}$ that acts as the identity on objects and that post-composes C-morphisms with the monad's unit. There also exists a functor $\mathsf{C}_{\mathbb{T}} \longrightarrow \mathsf{C}$ that acts like T on objects and that maps $\mathsf{C}_{\mathbb{T}}$-morphisms

$f : X \longrightarrow TY$ to C-morphisms $\mu_Y \cdot Tf : TX \longrightarrow TY$. Both functors form an adjunction,

$$\mathsf{C} \underset{\longleftarrow}{\overset{\longrightarrow}{\bot}} \mathsf{C}_{\mathbb{T}}$$

In other words, every monad gives rise to an adjunction. Conversely, every adjunction,

$$\mathsf{A} \underset{G}{\overset{F}{\bot}} \mathsf{B}$$

induces a monad $(GF, \eta, G\epsilon_F)$ where $\eta : \mathrm{Id} \longrightarrow GF$, $\epsilon : FG \longrightarrow \mathrm{Id}$ are, respectively, the unit and counit of the adjunction.

Example 1. *Consider a cartesian closed category* C *and an object* S *of that category. Then the adjunction,*

$$\mathsf{C} \underset{(-)^S}{\overset{(-\times S)}{\bot}} \mathsf{C}$$

gives rise to the state monad (which we studied earlier).

The fact that monads are born from adjunctions yields important results. For example, every set S and monad $\mathbb{T}$ over Set induce a combined monad defined by the composition of adjunctions,

$$\mathsf{Set} \underset{(-)^S}{\overset{(-\times S)}{\bot}} \mathsf{Set} \quad \bot \quad \mathsf{Set}_{\mathbb{T}}$$

where the adjunction on the right is the Kleisli adjunction of the monad $\mathbb{T}$.

Give an explicit definition of the monad that arises from the free-forgetful adjunction,

$$\mathsf{Set} \underset{U}{\overset{F}{\bot}} \mathsf{Mon}$$

where Mon is the category of monoids and monoid homomorphisms. Is it familiar to any monad that we studied before?

Recall that $\Delta : \mathsf{Set} \longrightarrow \mathsf{Set} \times \mathsf{Set}$ is the functor that duplicates a given object and that $(\times) : \mathsf{Set} \times \mathsf{Set} \longrightarrow \mathsf{Set}$ is the functor that sends a pair of objects into their product. Give an explicit definition of the monad that arises from the adjunction,

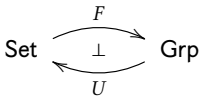
$$\mathsf{Set} \underset{(\times)}{\overset{\Delta}{\bot}} \mathsf{Set} \times \mathsf{Set}$$

Hint: This monad is useful for reading from a *one-bit memory*.

Let ComMon be the category of commutative monoids and monoid homomorphisms. Give an explicit definition of the monad that arises from the free-forgetful adjunction,

$$\mathsf{Set} \underset{U}{\overset{F}{\bot}} \mathsf{ComMon}$$

Let Grp be the category of groups and group homomorphisms. Give an explicit definition of the monad that arises from the free-forgetful adjunction,



Referências

[Lin66] Fred E. J. Linton. Some aspects of equational categories. In *Proceedings of the Conference on Categorical Algebra*, pages 84–94. Springer, 1966.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.