An introduction to that which shall not be named

Harley Eades III
Computer Science
Augusta University
heades@augusta.edu

September 2016

1 A quick overview from programming

Consider a C program with signature `int f(int)`. Now describe all possible computations this program can do. This is a difficult task, because this function could do a lot, like, prompt the user for input, send packets across the network, modify global state, and much more, but then eventually return an integer.

Now consider a purely functional programming language like Haskell [1] and list all the possible computations the function `f :: Int -> Int` can do. The list is a lot smaller. We know without a doubt that this function must take an integer as an input, and then do integer computations, and finally return an integer. No funny business went on inside this function. Thus, reasoning about pure programs is a lot easier.

However, a practical programmer might now be asking, "How do we get any real work done in a pure setting?". From stage left enters the monad. These allow for a programmer to annotate the return types of functions to indicate which side effects the function will use. For example, say we wanted `f` to use a global state of integers, then its type would be `f :: Int -> State [Int] Int` to indicate that `f` will take in an integer input, then during computation will use a global state consisting of a list of integers, but then eventually return an integer. Thus, the return type of `f` literally lists the side effects the function will use. Now while reasoning about programs we know exactly which side effects to consider.

In full generality a monad is a type constructor `m : * -> *` where `*` is the universe of types. Given a type `a` we call the type `m a` the type of computations returning values of type `a`. Thus, a function `f : a -> m b` is a function that takes in values of type `a`, and then returns a computation that will eventually return a value of type `b`.

Suppose we have `f :: a -> m b` and `g :: b -> m c`, and we wish to apply `g` to the value returned by `f`. This sounds perfectly reasonable, but ordinary composition `g.f` does not suffice, because the return type of `f` is not identical to the input type of `g`. Thus, we must come up with a new type of composition.

To accomplish this in Haskell we first need a new operator called *bind* which is denoted by `>>= :: m b -> (b -> m c) -> m c`. Then composition of `f` and `g` can be defined by `\x -> ((f x) >>= g) :: a -> m c`.

So we have a composition for functions whose type has the shape `a -> m b`, but any self respecting composition has an identity. This implies that we need some function `id :: a -> m a`, such that, `(\x -> (f x) >>= id)` = `f` and `(\x -> (id x) >>= g)` = `g`. This identity is denoted by `return :: a -> m a` in Haskell, and has to be taken as additional structure, because it cannot be defined in terms of bind.

Using `bind` and `return` in combination with products, sum types, and higher-order functions a large number of monads can be defined, but what are monads really?

2 What is a monad really?

3 A short history

[2]

4 Jumping inside a monad

4.1 The Kleisli category

4.2 The Eilenberg-Moore category

4.3 Why not an example from linear logic?

5 Monads are modular, right?

References

[1] The haskell programming language. Online: `http://www.haskell.org`.

[2] Eugenio Moggi. Computational lambda-calculus and monads. pages 14–23. IEEE Computer Society Press, 1988.