COMPUTING WITH MONADS

Directed Reading Project Report

Hyungki Kim

Mentor: Antoine Pouline

BACKGROUND

The usefulness and power of computation needs no debate. But even before the advent of the digital computer, mathematicians in the 1930s were trying to formally define a model for computation. That is, a series of mechanical steps that can be taken to complete a calculation. Turing, with his Turing machines, [1] Gödel with general recursive functions, [2] and Church with λ-calculus [3] independently came up with a model for computation. Later, these three models were proven to be equivalent, and we suspect that they capture the natural meaning of computation itself, in what is now known as the Church-Turing thesis.

Early on, computer scientists were mainly focused on how to translate the mathematical calculations they wanted to perform into machine instructions. Computers were modeled on the von Neumann architecture consisting of cpu and registers, and programming languages were mainly instructions for storing and fetching data from the registers, and executing code on the cpu. However, this quickly got complicated and hard to reason about. Thus, building on the efforts of Turing and Church, an effort was made to find mathematical models for programming. We wanted to be able to answer questions such as: When are two programs equivalent? What can we prove about the behavior and properties of programs? Can we formally guarantee the correctness of programs? Perhaps one of the most amenable models of programming to mathematical analysis is functional programming. [4]

Functional Programming

If we want to model functions in programs as functions in mathematics, we need to make sure they are well-defined. In functional programming (FP), such functions are called *pure* functions. Generally, a pure function $f : T_1 \to T_2$ maps from one type to another, where types can be

thought of as collections of possible values. In particular, f associates to every value of T_1, one, and only one value of T_2. If you have used a non-purely functional programming language before, you will realize that many of the "functions" you may have used don't possess this property. For example, in the following Python code,

```
x = 3
def f(n):
  return x + n

f(5) # returns 8
x = 1
f(5) # returns 6
```

the function `f` returns a different output give the same input in different circumstances. This additional context external to the input is called *state*. Thus, pure functions must be stateless. In addition, the purpose of certain functions is not to map inputs to outputs, but to have a *side effect*, that is, to change state. An example is the Python function `print`. A pure function must not have any side effects.

Functions in FP languages are usually *first class* or *higher-order* functions, which means functions themselves are treated as values or expressions. That is, functions can be the input to a function, and can be the output of a function. This allows for modeling function composition and recursively defined functions from mathematics.

Laziness

Infinite structures are often manipulated in mathematics without a second thought. However, this poses a problem for us if we want to represent infinite objects, such as the set of all natural numbers, as data. We do not have infinite storage nor infinite time to perform infinite computations. Fortunately, we usually don't need to use *all* the values. Rather, we can just compute up to the values we need. In standard

programming models, values are evaluated *eagerly*. That is, they are fully evaluated before being manipulated. In contrast, we can evaluate things *lazily*, and only evaluate values as we need them. For example, we can use generators in Python to make use of lazy evaluation to represent the natural numbers.

```
def N():    # the set of natural numbers
  n = 0
  while True:
    n += 1
    yield n

nats = N()
next(nats) # 1
next(nats) # 2
...
```

As a bonus, note that `N` is pure, whereas `next` is not.

HASKELL

Haskell is a purely functional, lazily evaluated programming language with algebraic data types, known for its expressive power. [5] In the midst of a revolution of many new ideas in the space, a committee got together to solidify these ideas and gave rise to the Haskell programming language. It acted as a testbed for researchers to implement their theoretical programming language ideas, but also served to experiment with the ideas of functional programming in building robust, reliable, and less error-prone software systems.

Effects

You may have realized that functional programming avoids side effects, but in order for software to actually be of use to us, side effects are necessary. What's the point in computing a solution to an equation

if the computer doesn't *output* the result? Or to take user inputs or write data. Generally, functional programming languages provide some kind of abstraction of the external environment, so that the user (programmer) can focus purely on the data going in and out of functions. The implementation of the language would handle performing the side effects behind the scenes. In Haskell, *streams* and *continuations* were used to provide I/O functionalities. However, there was a more elegant and general solution.

Monads

Monads are described as one of the most distinctive features of Haskell [5]. It allowed a unification and simplification of the different I/O abstractions, but was general enough to be applicable to other parts of the language as well.

Monads are a concept in Category Theory, first used by Moggi [6] to describe features of programming languages, like state management.

Definition (Monad). A Monad is an endofunctor $M : \mathrm{C} \to \mathrm{C}$ along with natural transformations

$$\eta : \mathrm{id}_{\mathrm{C}} \Rightarrow M$$
$$\mu : M^2 \Rightarrow M$$

such that

$$\mu \circ (\mathrm{id}_M \cdot \mu) = \mu \circ (\mu \cdot \mathrm{id}_M)$$
$$\mu \circ (\mathrm{id}_M \cdot \eta) = \mu \circ (\eta \cdot \mathrm{id}_M) = \mathrm{id}_M .$$

Note that η is also known as *unit* or *return*, and μ as *join*.

This is the minimal requirement for a monad. However, in Haskell, it's often more useful to define a monad with just unit along with *bind*

$$\mathrm{bind} : M(X) \times \hom(X, M(Y)) \to M(Y)$$

for $X, Y \in \mathrm{C}$ with

$$\text{bind} : (X, f) \mapsto f(X), \quad \text{and}$$
$$\mu_{M(Y)} : M(Y) \mapsto \text{bind}\,(Y, \text{id}_Y).$$

Bind allows us to easily compose monadic functions. Suppose we have a composition $A \xrightarrow{f} M(B)$ and $B \xrightarrow{g} M(C)$. We can't normally compose them since the domain doesn't match, but we can apply bind to compose them with $\text{bind}(f(A), g)$. Haskell provides bind as a composition operator `(>>=)` so that

```
bind (f x) g = (f x) >>= g
```

For our purposes, we consider the category Hask whose objects are the types in Haskell, and whose morphisms are Haskell functions. A monad generalizes the notion of *wrapping* a value with an additional context, be it state, effect, or any kind of computation. For example, take the `Maybe` monad

```
return :: a -> Maybe a
return x = Just x

bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind Nothing f = Nothing
bind (Just x) f = Just (f x)
```

which represents a computation that may or may not fail. This could be used when attempting to read a file which may not exist. Failure is represented by `Nothing`, and success with resulting value `x` by `Just x`. Showing that `return` and the resulting `join` are natural is left as an exercise.

Monadic I/O

The Haskell team realized that monads could be used to model in a pure way, not only states, but even general "computations" including performing side effects like I/O operations. For a function to be pure,

it cannot depend on anything external to its input. But how can we perform operations like taking input or printing output in a pure way? Every I/O operation in Haskell is modeled with the `IO` monad

```
type IO a = World -> (a, World)
```

That is, functions on `IO` take as their monadic context, the state of the entire external world, and returns an output along with a new state of the world. (one in which your monitor is now displaying the output, for example) Of course, Haskell doesn't allow a programmer to literally mold the world as they wish. It *is* a powerful language, but not that powerful. It merely allows the programmer to work as if this is the case, in a purely functional manner. The dirty details of actually perform the side effects is handled by the language under the hood. For example, here are the type signatures for the standard Haskell `IO` functions for getting input and printing output:

```
getLine :: IO String
putStr :: String -> IO ()
```

Note that `()` is the Unit type, which has just one value `()`.

Monoid in the Category of Endofunctors

Programmer's often have a hard time understanding monads. This led to a proliferation of tutorials on monads, leading to jokes such as that a rite of passage or a Haskell programmer is to write a monad tutorial, that "Monads are Burritos" regarding an oft-cited example of analogies used in these tutorials, a paper by Morehouse titled "Burritos for the Hungry Mathematician", [7] and the infamous quote that "a monad is a monoid in the category of endofunctors, what's the problem?" We will demonstrate the last statement, showing that a monad is really a categorification of the idea of a monoid. A monoid is a set with an associative binary operation, and an identity element. Naturally, the binary operation for categories would generalize to composition.

Definition (Monoidal Category). A *monoidal* category is a category C with a functor $\otimes : \mathrm{C} \times \mathrm{C} \to \mathrm{C}$ and a unit object 1 such that for all objects A, B, C and all morphisms f, g, h, we have:

- $A \otimes (B \otimes C) = (A \otimes B) \otimes C$
- $1 \otimes A = A$
- $A \otimes 1 = A$
- $f \otimes (g \otimes h) = (f \otimes g) \otimes h$
- $\mathrm{id}_1 \otimes f = f$
- $f \otimes \mathrm{id}_1 = f$

Definition (Monoid). An object M in a monoidal category $(\mathrm{C}, \otimes, 1)$ is a *monoid* object when there are two morphisms

$$\eta : 1 \to M$$
$$\mu : M \otimes M \to M$$

such that

$$\mu \circ (\mu \otimes \mathrm{id}_M) = \mu \circ (\mathrm{id}_M \otimes \mu)$$
$$\mu \circ (\eta \otimes \mathrm{id}_M) = \mu \circ (\mathrm{id}_M \otimes \eta) = \mathrm{id}_M .$$

By taking C to be the category of endofunctors, functor composition as the tensor product, and the identify functor id_C as the unit object, we can see that η and μ meet the conditions of the unit and join for monads. Thus monads are monoids in the category of endofunctors.

CONCLUSION

Composition plays a big role in programming, and especially in functional programming. Category theory provides an abstraction for composition, and monads provide a mathematical framework for handling state and side effects in a purely functional manner. This gives programmers an algebraic way to reason about programs.

However, theory has its practical limits. The category Hask, often used to model Haskell's type system, is not strictly a category. [8]

Undefined or bottom values $\perp$ break associativity and identity laws. Despite these limitations, monads and other abstractions remain useful, and provide structure for the otherwise messy world of computation.

BIBLIOGRAPHY

[1] Alan Turing, "On computable numbers, with an application to the Entscheidungsproblem," in Proc. Lond. Math. Soc, 1936.

[2] Kurt Gödel, "On undecidable propositions of formal mathematics systems," 1934.

[3] Alonzo Church, "An Unsolvable Problem of Elementary Number Theory," American Journal of Mathematics, 1936.

[4] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, 1978.

[5] Philip Wadler, Simon Peyton Jones, Paul Hudak, and John Hughes, "A History of Haskell:Being Lazy With Class," in Proc. of the 3rd ACM SIGPLAN, 2007.

[6] Eugenio Moggi, "Notions of Computation and Monads," Information and Computation, 1991.

[7] Ed Morehouse, "Burritos for the Hungry Mathematician," 2015. [Online]. Available: https://edwardmorehouse.github.io/silliness/burrito_monads.pdf

[8] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons, "Fast and loose reasoning is morally correct," Association for Computing Machinery, 2006.