CS150 APL: Monad

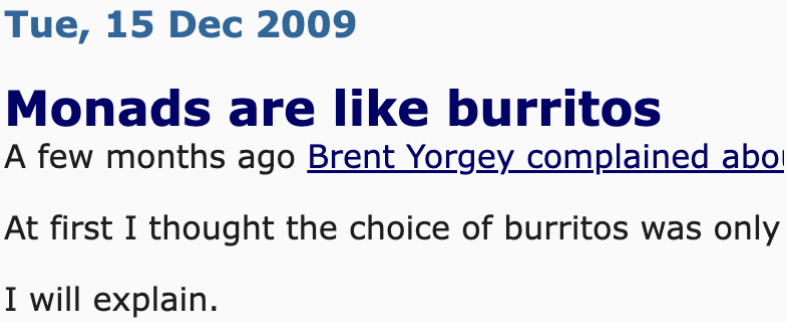
Guannan Wei
guannan.wei@tufts.edu

Nov 4, 2025

Tufts University

What is a Monad?

You may have seen this ...



Tue, 15 Dec 2009

Monads are like burritos

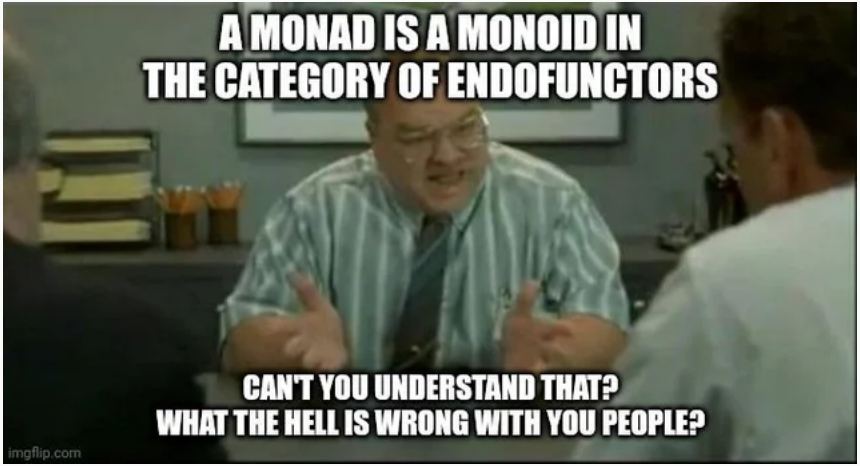
A few months ago Brent Yorgey complained abou

At first I thought the choice of burritos was only

I will explain.

What is a Monad?

and this ...



What is a Monad?

1990 - A committee formed by Simon Peyton-Jones, Paul Hudak, Philip Wadler, Ashton Kutcher, and People for the Ethical Treatment of Animals creates Haskell, a pure, non-strict, functional language. Haskell gets some resistance due to the complexity of using monads to control side effects. Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem?"

https://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

Today

A very pragmatic, programming-centric approach to make sense about monads.

Why care about monads?

- Originally designed to structure denotational semantics of languages (Moggi 1991)
- Representing effectful computation in pure languages
  - Easier to reason about (equational reasoning)
- Abstract over different kinds of computations
  - E.g., non-determinism, state, exceptions, I/O, etc.

Interpreter, Again

Define a simple arithmetic expression language:

```
enum Expr:
  case Num(n: Int)
  case Add(e1: Expr, e2: Expr)
  case Div(e1: Expr, e2: Expr)
```

Interpreter, Again

```
def eval0(e: Expr): Int =
  e match
    case Expr.Num(n) => n
    case Expr.Add(e1, e2) => eval0(e1) + eval0(e2)
    case Expr.Div(e1, e2) => eval0(e1) / eval0(e2)
```

Interpreter, Again

```
def eval0(e: Expr): Int =
  e match
    case Expr.Num(n) => n
    case Expr.Add(e1, e2) => eval0(e1) + eval0(e2)
    case Expr.Div(e1, e2) => eval0(e1) / eval0(e2)
```

What if we want to handle division by zero in our program logic?

```
eval0(Div(Num(1), Num(0))) // ???
```

Interpreter returning Option

Represent the normal return value and error as a "sum type":

```
enum Option[T]:
  case Value(v: T)
  case Error()
```

Define the interpreter:

```
def eval1(e: Expr): Option[Int] = ...
```

Interpreter returning Option

Then the interpreter can be modified as:

```
def eval1(e: Expr): Option[Int] =
  e match
    case Expr.Num(n) => Option.Value(n)
    case Expr.Add(e1, e2) =>
      eval1(e1) match
        case Option.Value(v1) =>
          eval1(e2) match
            case Option.Value(v2) => Option.Value(v1 + v2)
            case Option.Error()   => Option.Error()
        case Option.Error() => Option.Error()
```

Interpreter returning Option

and for the division case:

```
def eval1(e: Expr): Option[Int] =
  e match
    case Expr.Div(e1, e2) =>
      eval1(e1) match
        case Option.Value(v1) =>
          eval1(e2) match
            case Option.Value(v2) =>
              if v2 == 0 then Option.Error()
              else Option.Value(v1 / v2)
            case Option.Error() => Option.Error()
        case Option.Error() => Option.Error()
```

Interpreter counting the number of divisions

- Another variant of the interpreter, counting the number of divisions performed:
- `eval(Div(Num(4), Num(2)))` should return `(1, 2)`
- How should we change the interpreter?

Interpreter counting the number of divisions

- State-passing style:

```
type Count = Int
def eval2(e: Expr, n: Count): (Count, Int) =
  e match
    case Expr.Num(v) => (n, v)
    case Expr.Add(e1, e2) =>
      val (n1, v1) = eval2(e1, n)
      val (n2, v2) = eval2(e2, n1)
      (n2, v1 + v2)
    case Expr.Div(e1, e2) =>
      val (n1, v1) = eval2(e1, n)
      val (n2, v2) = eval2(e2, n1)
      (n2 + 1, v1 / v2)
```

Monad

We have seen

- ordinary interpreter `eval0`
- interpreter with error handling `eval1`
- interpreter with state `eval2`

Can we abstract out the common pattern here?

Monad Type

- `M[_]: * -> *` is a type constructor that takes a type and produces a new type.
- `unit: A -> M[A]` (also called `return` or `pure`)
- `bind: M[A] -> (A -> M[B]) -> M[B]` (also called monadic application or `flatMap`)

Monad Type

- `M[_]: * -> *` is a type constructor that takes a type and produces a new type.
- `unit: A -> M[A]` (also called `return` or `pure`)
- `bind: M[A] -> (A -> M[B]) -> M[B]` (also called monadic application or `flatMap`)

- Algebraic laws:
  - left identity: `bind unit(a) f = f(a)`
  - right identity: `bind m (x => unit(x)) = m`
  - associativity: `bind (bind m f) g = bind m (x => bind f(x) g)`

Monad Type

- `M[_]: * -> *` is a type constructor that takes a type and produces a new type.
- `unit: A -> M[A]` (also called `return` or `pure`)
- `bind: M[A] -> (A -> M[B]) -> M[B]` (also called monadic application or `flatMap`)
- Algebraic laws:
  - left identity: `bind unit(a) f = f(a)`
  - right identity: `bind m (x => unit(x)) = m`
  - associativity: `bind (bind m f) g = bind m (x => bind f(x) g)`
- `<M, unit, bind>` is called a Kleisli triple.

Monadic Interpreter

Declare the monad type and operations:

```
type M[T]
def unit[T](v: T): M[T] = ???
def bind[T, U](m: M[T], f: T => M[U]): M[U] = ???
def divM(v1: Int, v2: Int): M[Int] = ???
```

Monadic Interpreter

```
def evalM(e: Expr): M[Int] =
  e match
    case Expr.Num(v) => unit(v)
    case Expr.Add(e1, e2) =>
      bind(evalM(e1), v1 =>
        bind(evalM(e2), v2 =>
          unit(v1 + v2)))
    case Expr.Div(e1, e2) =>
      bind(evalM(e1), v1 =>
        bind(evalM(e2), v2 =>
          divM(v1, v2)))
```

Recovering the previous interpreters: Id Monad

```
type M[T] = T
def unit[T](v: T): M[T] = v
def bind[T, U](m: M[T], f: T => M[U]): M[U] = f(m)
def divM(v1: Int, v2: Int): M[Int] = v1 / v2
```

The `evalM` now behaves like `eval0`.

Recovering the previous interpreters: Option Monad

```
type M[T] = Option[T]
def unit[T](v: T): M[T] = Option.Value(v)
def bind[T, U](m: M[T], f: T => M[U]): M[U] =
  m match
    case Option.Value(v) => f(v)
    case Option.Error()  => Option.Error()
def divM(v1: Int, v2: Int): M[Int] =
  if v2 == 0 then Option.Error()
  else Option.Value(v1 / v2)
```

The `evalM` now behaves like `eval1`.

Recovering the previous interpreters: State Monad

```
type M[T] = Count => (Count, T)
def unit[T](v: T): M[T] = n => (n, v)
def bind[T, U](m: M[T], f: T => M[U]): M[U] =
  n =>
    val (n1, v) = m(n)
    f(v)(n1)
def divM(v1: Int, v2: Int): M[Int] = n => (n + 1, v1 / v2)
```

The `evalM` now behaves like `eval2`.

Monad Examples

- `M[T] = T` is the identity monad
- `M[T] = T + None` is the option monad
- `M[T] = S => (S, T)` is the state monad
- `M[T] = R => T` is the reader/environment monad
- `M[T] = List[T]` is the list/nondeterminism monad
- `M[T] = (T => R) => R` is the continuation monad
- …

Monadic Interpreter

It would be nice to write this monadic interpreter in more natural syntax...

```
def evalM(e: Expr): M[Int] =
  e match
    case Expr.Num(v) => unit(v)
    case Expr.Add(e1, e2) =>
      bind(evalM(e1), v1 =>
        bind(evalM(e2), v2 =>
          unit(v1 + v2)))
    case Expr.Div(e1, e2) =>
      bind(evalM(e1), v1 =>
        bind(evalM(e2), v2 =>
          divM(v1, v2)))
```

Monadic Interpreter

Haskell adopts the do-notation in 1998 for this:

```
evalM :: Monad M => Expr -> M Int
evalM (Num v) = return v
evalM (Add e1 e2) = do
  v1 <- evalM e1
  v2 <- evalM e2
  return (v1 + v2)
evalM (Div e1 e2) = do
  v1 <- evalM e1
  v2 <- evalM e2
  divM v1 v2
```

`do x <- m` is syntactic sugar for `bind m (\x -> ...)`

Monadic Interpreter with better syntax in Scala

Define some type classes to enable better syntax:

```
trait Functor[F[_]]:
  extension [A, B](x: F[A])
    def map(f: A => B): F[B]

trait Monad[M[_]] extends Functor[M]:
  def pure[A](x: A): M[A]
  extension [A, B](m: M[A])
    def flatMap(f: A => M[B]): M[B]
    def map(f: A => B): M[B] = m.flatMap(f.andThen(pure))
```

Monadic Interpreter with better syntax in Scala

Scala recognizes for-comprehension syntax with `map` and `flatMap` (ie `bind`) methods:

```
def evalM[M[_]](e: Expr)(using m: Monad[M]): M[Int] = e match
  case Expr.Num(v) => m.pure(v)
  case Expr.Add(e1, e2) => ...
  case Expr.Div(e1, e2) =>
    for {
      v1 <- evalM(e1)
      v2 <- evalM(e2)
      r <- divM(v1, v2)
    } yield r
```

Summary

- Monads provide a powerful abstraction for structuring computations with effects.
- They allow us to write generic interpreters that can be instantiated with different effects.
- Paper: Monads for functional programming, Philip Wadler, 1992