Week 14a: Monads

CS-214 Software Construction

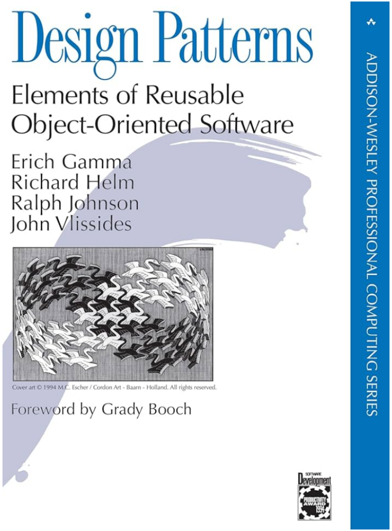
Outline

- What are Monads
- Monad Laws
- For Comprehensions
- List and Exception Monads
- Monadic Translation: val to for
- State Monad

What are Monads

CS-214 Software Construction

Monad is a Design Pattern for Functional Programmers



Brief History of Monads

Monad word was used by Pythagoreans, later philosophers, then by Leibniz

- this historical use of the word is not relevant for this lecture

Directly relevant concepts:

- used into Category theory from 1950ies
- used to translate programs to math, by Eugenio Moggi in 1989; Category theory can represent function-like concepts, such as functions with side effects
- Wadler in 1990 observed that list comprehensions (Scala's `for` comprehensions) can be explained and generalized in terms of monads
- Peyton Jones and Wadler in 1993 argued to use monads for I/O in pure languages leading to their central role in Haskell. Later: `do` notation in Haskell
- F^* verifier (`https://www.fstar-lang.org/`) uses monads to reduce reasoning about effects to functions, without the need to use do notation

Monads are about flatMap

Unlike Haskell, Scala permits effects (e.g. mutable fields, exceptions), so no need to model them using monads (programs become less readable in monadic style)

Most common use of monads in Scala is for collection types that support `flatMap`

- `Seq[T]` (`List[T]`, `Vector[T]`), `Option[T]`, `Try[T]` ...

Remember `flatMap` on `List[T]`:

```
extension[T](m: List[T])
  def flatMap(f: T => List[U]): List[U] =
    m match
      case Nil    => Nil
      case h :: t => f(h) ++ t.flatMap(f)

// List(t1, t2, t3)  ~~.flatMap(f)~~>  f(t1) ++ f(t2) ++ f(t3)
```

We first list elements such as `t1`, then function `f` (`feed ‘t1‘ to ‘f‘`)

Monad Definition

Informally, monad is a collection-like generic class, `M[A]`, with a `flatMap`.

Monad is given by:

1. a generic type `M[A]` (generalizing `List[A]`, `Option[A]`, etc.)
2. a function `unit[A]: A => M[A]` (generalizing `a => List(a)`)
3. `flatMap[B]` method on `m : M[A]` that applies `f : A => M[B]`, obtaining `M[B]`:

```
m.flatMap(f): M[B]
```

where `unit` and `flatMap` satisfy certain the laws that we will discuss shortly.

Other names for `flatMap` include: `bind`, $\star$, `>>=`

Identity Monad (Mostly Useless)

```scala
type Id[A] = A

def unit[A](a: A): Id[A] = a

extension(a: Id[A])
  def flatMap[B](f: A => Id[B]): Id[B] =
    f(a)
```

`a.flatMap(f) = f(a)` so it's just function application with argument order swapped

Option as a Monad

```scala
sealed trait Option[+A]:
case object None extends Option[Nothing]
final case class Some[+A](content: A) extends Option[A]

def unit[A](a: A): Option[A] = Some(a)

extension(ma: Option[A])
  def flatMap[B](f: A => Option[B]): Option[B] =
    ma match
      case None => None
      case Some(a) => f(a)
```

Note: `ma.flatMap(f)` serves the purpose of `f(ma)` but propagates `None`

Monad Laws

CS-214 Software Construction

Simpler Structure: Monoid and Its Laws

Monoid is an algebraic structure with a binary operation `op` and an element `u`

If `op` is a method, the laws of monoid are:

```
u.op(x) === x                        // u is left unit
x.op(u) === x                        // u is right unit
x.op(y).op(z) === x.op(y.op(z))      // associativity
```

When we use infix operator, e.g., *, then they look more familiar:

```
u * x === x
x * u === x
(x * y) * z === x * (y * z)
```

For *monads*, the operator `flatMap` takes a *function* as its right operand.

Moreover, there is not just one unit u, but a unit(x) for each `x`

Monoid vs Monad Laws

Monoid laws:

```
u.op(x) === x                        // u is left unit
x.op(u) === x                        // u is right unit
x.op(y).op(z) === x.op(y.op(z))      // associativity
```

Monad laws are similar, but pass a value from left to function on the right of `flatMap`:

For all

```
a: A     ma: M[A]     ab: A => M[B]     bc: B => M[C]
```

the following must hold:

```
unit(a).flatMap(ab) === ab(a)                                           (l. unit)
ma.flatMap(unit)     === ma                                             (r. unit)
ma.flatMap(ab).flatMap(bc) === ma.flatMap((a:A) => ab(a).flatMap(bc))   (assoc.)
```

Monad Associativity Explained (Option as Example)

- `ma: Option[A]`
- `ab: A => Option[B]`
- `bc: B => Option[C]`

Then we can get `Option[C]` in two ways; law says the result is same:

1. apply first `ab` and then `bc`

```
(ma.flatMap(ab)).flatMap(bc)       // first parenthesis not needed in Scala
```

2. compose `ac` and `ma` and apply it to `ma`:

```
val ac: A => Option[C] =
  (a: A) => ab(a).flatMap(bc)
ma.flatMap(ac)                     // can replace ac with its definition above
```

In both cases, the result is applying `ab` and then `bc`, propagating `None`

Monad Associativity with New-Line Syntax

- `ma: Option[A]`
- `ab: A => Option[B]`
- `bc: B => Option[C]`

The law:

```
ma.flatMap(ab).flatMap(bc) === ma.flatMap(a => ab(a).flatMap(bc))
```

with `bc` function expanded into `b => bc(b)` on right side, becomes:

```
    ma.flatMap(ab).flatMap(bc)
=== ma.flatMap: a =>
      ab(a).flatMap: b =>
        bc(b)
```

Doing flatMap one by one is same as binding values

Remark on Extensionality

We will generally assume that two functions,

```
f: A => B
g: A => B
```

are mathematically equal if and only if, for all values `x: A`,

```
f(x) === g(x)
```

Consequence: if `f` is a function, then

```
    (y => f(y))
===
    f
```

Indeed, applying both sides to some value `v` gives `f(v)`

`for` Comprehensions

CS-214 Software Construction

`for` Comprehensions

Instead of writing, e.g.

```
ma.flatMap: a =>
  ab(a).flatMap: b =>
     bc(b)
```

we can write, gaining or losing essentially nothing:

```
for
  a <- ma
  b <- ab(a)
  c <- bc(b)
yield c
```

Compiler will translate the `for` expression into `flatMap` calls for us.

map on a Monad

```scala
sealed trait Option[+T]:
  def flatMap[U](f: T => Option[U]): Option[U] = ...

  def map[U](f: T => U): Option[U] =
    flatMap((x:T) => unit(f(x)))        // definition for map in a Monad
```

Right unit law

```scala
m.flatMap(unit) === m
```

is (by expanding `map` definition) equivalent to:

```scala
m.map(x => x) === m
```

as

```scala
m.flatMap(x => unit(x)) === m.flatMap(unit)
```

Translation uses flatMap + map instead of flatMap + unit

Expression

```
for a <- ma
    b <- ab(a)
    c <- bc(b)
yield d(c)
```

in Scala means

```
ma.flatMap: a =>
  ab(a).flatMap: b =>
    bc(b).map: c =>
      d(c)
```

(Unlike Haskell, `yield` is a keyword and not a `unit`.)

Monad Unit Laws Using `for` Comprehensions

Right unit law:

```
    for a <- ma
    yield a
===
    ma
```

Left unit law (given right unit law):

```
    for a <- unit(x)
        b <- f(a)
    yield b
===
    f(x)
```

Associativity Law Using `for` Comprehensions

```
    for b <- (for a <- ma
                  b1 <- ab(a)
              yield b1)
        c <- bc(b)
    yield c
===
    for a <- ma
        b <- ab(a)
        c <- bc(b)
    yield c
```

Exercise: Define Flatten: M[M[A]] => M[A]

```scala
extension[A](mma: Option[Option[A]])
  def flatten: Option[A] =
    for ma <- mma
        a <- ma
    yield a

 // that is,  mma.flatMap(ma => ma.map(a => a))
          === mma.flatMap(ma => ma)
```

Conversely, given `map` and `flatten` we can define `flatMap`:

```scala
extension[A](ma: Option[A])
  def flatMap[B](f: A => Option[B]): Option[B] =
    ma.map(f).flatten
```

One could alternatively define monads using `map` and `flatten`

List and Exception as Monads

CS-214 Software Construction

Two Directions to Generalize Option

We view expression of Option[T] as a computation that:

1. returns one or more results - generalization: return any number of results using List[T], Seq[T]
2. succeeds or fails with None - generalization: indicate reason for failure using Try[T]

Being a monad is just a common interface, but each monad is useful because it has its own intrinsic operations that make it useful.

List as a Monad

```scala
sealed abstract class List[+A]:
  def flatMap[B](f: A => List[B]): List[B] =
    this match
      case Nil => Nil
      case Cons(a, as) => f(a) ++ as.flatMap[B](f)

  def ++[B >: A](that: List[B]): List[B] =
    this match
      case Nil => that
      case Cons(a, as) => Cons(a, as ++ that)

case object Nil extends List[Nothing]
case class Cons[+A](first: A, next: List[A]) extends List[A]

def unit[A](x: A): List[A] = Cons(x, Nil)
```

Uses of List (and, similarly, LazyList) as a Monad

Can be viewed as representing computation with multiple results

- expressing search problems
- expressing generation of values for testing as in ScalaCheck

Useful additional operations (beyond flatMap):

- concatate lists: join two `ma1, ma2: M[A]`
- filter: only permit values that satisfy a condition

Monad with "zero" and "plus"

Try as Monad for Exceptions (see Week 11)

Suppose we are in a pure language and wish to represent exceptions

```
sealed abstract class Try[+A]:
  def flatMap[B](f: A => Try[B]): Try[B] =
    this match
      case Success(v) => f(v)
      case f @ Failure(_) => f

case class Success[+A](value: A) extends Try[A]
case class Failure(msg: String) extends Try[Nothing]

def unit[A](x: A): Try[A] = Success(x)
```

Additional Operations: Throw and Catch

```
def THROW[A](msg: String): Try[A] = Failure(msg)

extension[A](t: Try[A])
  def CATCH(handler: String => Try[A]): Try[A] =
    t match
      case Success(v) => t
      case Failure(msg) => handler(msg)
```

Illustration with Division

```
extension[T](x: Try[Double])
  def +(y: Try[Double]) =
    x.flatMap: xd =>
      y.flatMap: yd =>
        unit(xd + yd)

  def /(y: Try[Double]): Try[Double] =
      x.flatMap: xd =>
        y.flatMap: yd =>
          if Math.abs(yd) <= Double.MinPositiveValue then THROW("Division by zero")
          else unit(xd / yd)

def harmonicMean(x: Try[Double], y: Try[Double]) =
  (one / (one / x + one / y)).CATCH: msg =>
    unit(0.0)
```

Future Core API (Week 12)

Future provides convenient high-level transformation operations.

```
trait Future[+A]:
  def onComplete(k: Try[A] => Unit): Unit

  // transform successful results:
  def flatMap[B](f: A => Future[B]): Future[B]             <====
  def map[B](f: A => B): Future[B]                         <====
  def zip[B](fb: Future[B]): Future[(A, B)]

  // transform failures
  def recover(f: Exception => A): Future[A]
  def recoverWith(f: Exception => Future[A]): Future[A]
```

`flatMap` Operation on `Future`

```scala
trait Future[+A]:
  ...
  def flatMap[B](f: A => Future[B]): Future[B]      // f itself can fail
```

- Transforms a successful `Future[A]` into a `Future[B]` by applying a function `f: A => Future[B]` after the `Future[A]` has completed
- Returns a failed `Future[B]` if the former `Future[A]` failed or if the `Future[B]` resulting from the application of the function `f` failed.

```scala
  def map[B](f: A => B): Future[B]          // f cannot fail, must give B
```

- Transforms a successful `Future[A]` into a `Future[B]` by applying a function `f: A => B` after the `Future[A]` has completed
- Automatically propagates the failure of the former `Future[A]` (if any), to the resulting `Future[B]`

Parsers (Syntax Analyzers)

A parser is a function that transforms

- a linear sequence (String, text file) into
- a tree representation (e.g. expression tree, JSON tree, XML tree).

Parsers are used in compilers, interpreters, web browsers, network software, Linux and Windows operating system kernel, text editors and IDEs,...

Monadic Parser Combinators concisely implement parsers using monads

```
// parser consumes some string, produces tree of type A, returns leftover string
// (it can return a lazy list of possibilities, empty list is syntax error)
case class Parser[A](runOn: String => LazyList[(A,String)])
  ...
  def flatMap(f: A => Parser[B]): Parser[B]
```

More about parsing in: Computer Language Processing (CS-320)

Monadic Translation: val to for

CS-214 Software Construction

We Have Seen Addition of Try[Double]

We had `+ : (Double, Double) => Double`

wanted `(Try[Double], Try[Double]) => Try[Double]`

```
extension[T](x: Try[Double])
  def +(y: Try[Double]): Try[Double] =
    x.flatMap: xd =>
      y.flatMap: yd =>
        unit(xd + yd)
// i.e.
    for xd <- x
        yd <- y
    yield x + y
```

It is not specific to `Try`. Works analogously if you replace `Try` with, e.g., `List`

There is a general way to transform pure computation into one that uses a given monad

Call-by-Value Monadic Translation: replace val with for

Consider an expression such as

```
e(f1(x), f2(x))
```

We can break it down into individual steps using `val`-s:

```
val x1 = f1(x)
val x2 = f2(x)
val res = e(x1, x2)
res
```

To represent this computation using monads, we write simply:

```
for x1 <- f1(x)         // f1 now returns a monad
    x2 <- f2(x)
    res <- e(x1, x2)
yield res
```

Monadic Translation of Functions

Consider `f: A => B`, `x: A`, so: `f(x): B`

Translation replaces `f(x)` function application with `x'.flatMap(f')`, so we should have

```
f': A => M[B]              // monad added only to function result
x': M[A]
x'.flatMap(f'): M[B]
```

Hence, anonymous function

```
(x:A) => (e:B)
```

should become:

```
(x:A) => (e': M[B])
```

Example: Translation of foldLeft to use Try

```
def foldLeft[A,B](lst: List[A], z: B, op: (B,A) => B): B =
  lst match
    case List() => z
    case x :: xs =>
      val z1 = op(z, x)
      foldLeft(xs, z1, op)

def foldLeftM[A,B](lst: List[A], z: B, op: (B,A) => Try[B]): Try[B] =
  lst match
    case List() => Try(z)   // unit(z)
    case x :: xs =>
      op(z,x).flatMap: z1 =>
        foldLeftM(xs, z1, op)
```

State Monad

CS-214 Software Construction

Recall Week 10: Making Arguments and Results Explicit

```
type State = Int
var s: State = 2
{s = s + 2; s} + {s = s - 1; 10*s}     // What is the result?

          4  +              30
       == 34
```

A block of code returns a value

But it also has an effect: it depends on state s and it changes s

The meaning of a block returning value of type `A` is a function `f: State => (A,State)`

- when we execute code in state `s` then if `f(s)` is `(a,s')` it means that the code:
  - returns `a`
  - changes the state to `s'`

Composing Blocks of Code with Plus

```
{s = s + 2; s} + {s = s - 1; 10*s}

      b1                 b2
```

```scala
type State = Int
type Block[A] = State => (A, State)
def addBlockValues[T](b1: Block[Int], b2: Block[Int]): Block[Int] =
  (s0: State) =>
    val (r1:Int, s1:State) = b1(s0)
    val (r2:Int, s2:State) = b2(s1)
    (r1 + r2, s2)
```

flatMap for State Monad: Pass Value of Block to a Function

```
type Block[A] = State => (A,State)
extension[A](b: Block[A])
  def flatMap[B](f: A => Block[B]): Block[B] =
    (s0: State) =>
      val (a:A, s1:State) = b(s0)
      f(a)(s1)
```

`Block[T]` is a state monad, with state Int.

State in the state monad can be arbitrary, so we make it a parameter: `StateMonad[S,A]`

```
type StateMonad[S,A] = S => (A, S)
```

What is the unit?

```
def unit[S,A](a: A): StateMonad[S,A] =  (s:S) => (a, s)
```

Example of a Special Operation for State Monad

```scala
def getInc: StateMonad[Int,Int] =
  (s:Int) => (s, s + 1)
```

Fundamental operations:

```scala
def read[S]: StateMonad[S,S] = (s:S) => (s, s)
```

```scala
def write[S](s0:S): StateMonad[S,Unit] = (s:S) => ((), s0)
```

Exercise: write `getInc` using `read` and `write` (and `flatMap` / `for`)

Addition of Side-Effecting Expressions

```
{s = s + 2; s} + {s = s - 1; 10*s}
```

the plus in `...} + {...`becomes

```
def add[T](b1: StateMonad[Int, Int],
           b2: StateMonad[Int,Int]): StateMonad[Int, Int] =
  b1.flatMap: r1 =>
    b2.flatMap: r2 =>
      unit(r1 + r2)
```

Same structure of code as the addition that takes Try values!

Example: Renaming

```
def rename(lst: List[String]): StateMonad[Int,List[String]] =
  lst match
    case List() => unit[Int,List[String]](List())
    case x :: xs =>
      for count <- getInc
          xs1 <- rename(xs)
      yield f"${x}_${count}" :: xs1

def run[S,A](st: StateMonad[S,A])(s0: S): A =
  val (a, s1) = st(s0)
  a

val lst = List("a", "b", "a")
val m = rename(lst)
println(run(m)(10))  // List(a_10, b_11, a_12)
```

To Explore More

See the monad exercises this week!

Using a common interface with flatMap makes code more modular

However:

- code needs to be flattened, looks less readable
- we may lose parallelism
- combining different types of monads becomes messy

Be aware of monads, but beware of using monads too much.