Monads for normal people!

Dustin Getz
@dustingetz

https://github.com/dustingetz/monadic-interpreter
https://github.com/dustingetz/pymonads

intended audience

- coders
- who are comfortable with lambdas
- who learn by example

goals

- how do monads work
- how do monads help
- are monads useful IRL?
- especially in enterprise?
- where do they fall short and what's next

large codebases are complex

- Spring, EJB, AspectJ, DI, AOP
- Common goal: make code look like business logic
- (to varying degrees of success)

Aspect Oriented Programming

From Wikipedia, the free encyclopedia

In computing, **aspect-oriented programming** (**AOP**) is a programming paradigm which aims to increase modularity by allowing the separation of cross-cutting concerns.

Typically, an aspect is *scattered* or *tangled* as code, making it harder to understand and maintain. It is scattered by virtue of the function (such as logging) being spread over a number of unrelated functions that might use *its* function, possibly in entirely unrelated systems, different source languages, etc. That means to change logging can require modifying all affected modules. Aspects become tangled not only with the mainline function of the systems in which they are expressed but also with each other. That means changing one concern entails understanding all the tangled concerns or having some means by which the effect of changes can be inferred.

Lots of code to follow

- Pay attention to how the functions compose

a bank API

```
def get_account(person):
    if person.name == "Alice": return 1
    elif person.name == "Bob": return 2
    else: return None

def get_balance(account):
    if account.id == 1: return 1000000
    elif account.id == 2: return 75000
    else: return None

def get_qualified_amount(balance):
    if balance.cash > 200000: return balance.cash
    else: return None
```

(shoutout to Tumult on hackernews for the 'bank' example)

what we want to write

```
def get_loan(name):
    account = get_account(name)
    balance = get_balance(account)
    loan = get_qualified_amount(balance)
    return loan
```

My boss could write this code

```
get_qualified_amount( get_balance( get_account( alice )))

alice | get_account | get_balance | get_qualified_amount

(-> get_account get_balance get_qualified_amount)(alice)

alice.get_account().get_balance().get_qualified_amount()
```

i love NullPointerExceptions

```
def get_account(person):
    if person.name == "Alice": return 1
    elif person.name == "Bob": return 2
    else: return None


>>> get_account(None)
AttributeError: 'NoneType' object has no attribute 'id'
```

what the prod code looks like :(

```
def get_loan(person):
    account = get_account(person)
    if not account:
        return None
    balance = get_balance(account)
    if not balance:
        return None
    loan = get_qualified_amount(balance)
    return loan
```

factor! abstract! happy!

```
def bind(v, f):
    if (v):
        return f(v)
    else:
        return None
```

factor! abstract! happy!

```
def bind(v, f):
    if (v):                        # v == alice
        return f(v)                # get_account(alice)
    else:
        return None

>>> alice = Person(...)
>>> bind(alice, get_account)
1
```

factor! abstract! happy!

```
def bind(v, f):
    if (v):
        return f(v)
    else:                       # v == None
        return None             # don't call f

>>> bind(None, get_account)
None
```

the code we really want to write

```
def bind(v,f): return f(v) if v else None

def get_loan(name):
    m_account = get_account(name)
    m_balance = bind(m_account, get_balance)
    m_loan =    bind(m_balance, get_qualified_amount)
    return m_loan


>>> alice = Person(...)
>>> get_loan(alice)
100000
>>> get_loan(None)
None
```

or more succinctly

```
def bind(v,f): return f(v) if v else None

def m_pipe(val, fns):
    m_val = val
    for f in fns:
        m_val = bind(m_val, f)
    return m_val


>>> fns = [get_account, get_balance, get_qualified_amount]
>>> m_pipe(alice, fns)
1000000
>>> m_pipe(dustin, fns)
None
```

big picture goal

- make the code look like the business logic
- "good clojure programmers write a language to write their program in" -- DSLs
- build a language to build your business logic
- add features without changing your business logic

Great things are made of little things.

add a feature to our API

```
def get_account(person):
   if person.name == "Alice": return (1, None)
   elif person.name == "Bob": return (2, None)
   else: return (None, "No acct for '%s'" % person.name)
```

```
>>> get_account(alice)
(1, None)

>>> get_account(dustin)
(None, "No acct for 'Dustin'")
```

```
def get_balance(account):
  if account.id == 1: return (1000000, None)
  elif account.id == 2: return (75000, None)
  else: return (None, "No balance, acct %s" %
                          account.id)

def get_qualified_amount(balance):
  if balance.cash > 200000: return (balance, None)
  else: return (None, "Insufficient funds: %s" %
                          balance.cash)
```

```
def bind(mval, mf):
    value = mval[0]
    error = mval[1]
    if not error:
        return mf(value)
    else:
        return mval
```

```
def bind(mval, mf):             # mval == (1, None)
    value = mval[0]             # value == 1
    error = mval[1]             # error == None
    if not error:
        return mf(value)        # mf(1)
    else:
        return mval
```

```
>>> mval = (1, None)
>>> bind(mval, get_balance)
(100000, None)
```

```
def bind(mval, mf):        # mval == (None, "insuf funds")
    value = mval[0]        # value == None
    error = mval[1         # error == "insuf funds"
    if not error:
        return mf(value)
    else:
        return mval        # short circuit
```

```
>>> mval = (None, "insuf funds")
>>> bind(mval, get_balance)
(None, "insuf funds")
```

Business logic didn't change

```
# error monad
def bind(mv, mf): mf(mv[0]) if mv[0] else mv
def unit(v): return (v, None)

def m_pipe(val, *fns):
    m_val = unit(val)
    for f in fns:
        m_val = bind(m_val, f)
    return m_val


>>> m_pipe(alice, get_account, get_balance,
             get_qualified_amount)
(1000000, None)
```

Business logic didn't change

```
# error monad
def bind(mv, mf): mf(mv[0]) if mv[0] else mv
def unit(v): return (v, None)

def m_chain(*fns): ...

>>> get_loan = m_chain(get_account, get_balance,
                       get_qualified_amount)
<fn that composes the fns in order>

>>> get_loan(alice)
(1000000, None)
>>> get_loan(dustin)
(None, "insuf funds")
```

Business logic didn't change

```
get_loan = m_chain(get_account, get_balance,
                   get_qualified_amount)


>>> map(get_loan, [alice, dustin, bob])
[(1000000, None), (None, "insuf funds"), (75000, None)]


>>> mmap(get_loan, [alice, bob])
([1000000, 75000], None)

>>> mmap(get_loan, [alice, bob, dustin])
(None, "insuf funds")
```

here be monads

```
# maybe monad
def bind(mv, mf): return mf(mv) if mv else None
def unit(v): return v

# error monad
def bind(mv, mf): mf(mv[0]) if mv[0] else mv
def unit(v): return (v, None)
```

monad comprehensions

```
# error monad
def bind(mv, mf): mf(mv[0]) if mv[0] else mv
def unit(v): return (v, None)

>>> mv1 = (7, None)
>>> mv2 = bind( mv1,
                lambda x: unit(x))
(7, None)

>>> mv3 = bind( mv2,
                lambda x: unit(x+1))
(8, None)
```

monad comprehensions

```
# error monad
def bind(mv, mf): mf(mv[0]) if mv[0] else mv
def unit(v): return (v, None)

>>> mv1 = (7, None); mv2 = (3, None)
>>> mv3 = bind( mv1,
                lambda x: bind( mv2,
                                lambda y: unit(x + y)))
(10, None)

>>> mv4 = (None, “error”)
>>> bind( mv3,
          lambda x: bind( mv4,
                          lambda y: unit(x + y)))
(None, “error”)
```

the simplest monad

```
# identity monad
def bind(mv,mf): return mf(mv)
def unit(v): return v


>>> bind( 7,
          lambda x: bind( 3,
                          lambda y: unit(x + y)))
10
```

the simplest monad

```
# identity monad
def bind(mv,mf): return mf(mv)
def unit(v): return v


>>> bind( 7,    lambda x:
    bind( 3,    lambda y:
          unit(x + y)))
10

repl> (let [x 7
            y 3]
         x + y)
10
```

Lists are a monad!

```
>>> [(x,y) for x in ranks for y in files]

# list monad
def unit(v): return [v]
def bind(mv,mf): return flatten(map(mf, mv))

>>> ranks = list("abcdefgh")
>>> files = list("12345678")
>>> bind( ranks,    lambda rank:
    bind( files,    lambda file:
          unit((rank, file))))
repl> (for [rank (seq “abcdefgh”)
            file (seq “12345678”)]
         [rank, file])
```

monads are...

- a design pattern for composing functions that have incompatible types, but can still logically define composition
- by overriding function application
- to increase modularity and manage accidental complexity

here comes the fun stuff

- introduce a few harder monads
- combine monads to build...
  - a Clojure parsing DSL
  - a Python lisp interpreter

writer monad

```
def unit(v): return (v, [])
def bind(mv, mf): …

def addOne(x):
    val = x+1
    logmsg = "x+1==%s" % val
    return (val, [logmsg])

>>> mv = addOne(7)
(8, ['x+1==8'])
>>> m_chain(addOne, addOne, addOne)(mv)
(11, ['x+1==9', 'x+1==10', 'x+1==11'])
```

writer monad

```
def unit(v): return (v, [])
def bind(mv, mf): …

def addOne(x):
    val = x+1
    logmsg = "x+1==%s" % val
    return (val, [logmsg])

>>> addThreeLogged = m_chain(addOne, addOne, addOne)
>>> map(addThreeLogged, [10,20,30])
[(13, ['x+1==11', 'x+1==12', 'x+1==13']),
 (23, ['x+1==21', 'x+1==22', 'x+1==23']),
 (33, ['x+1==31', 'x+1==32', 'x+1==33'])]
```

writer monad

```
def unit(v): return (v, [])
def bind(mv, mf): …

def addOne(x):
    val = x+1
    logmsg = "x+1==%s" % val
    return (val, [logmsg])

>>> mmap(addOne, [1,2,3])
   ([2, 3, 4], ['x+1==2', 'x+1==3', 'x+1==4'])
```

reader monad

```
def unit(v): return lambda env: v
def bind(mv, mf): …

def read(key):
    def _(env):
        return env[key]
    return _

>>> mv = read('a')
<a function of env>
>>> mv({'a': 7})
7
>>> mv({'a': 10, 'b': 3})
10
```

reader monad

```
def unit(v): return lambda env: v
def bind(mv, mf): …

def read(key):
    def _(env):
        return env[key]
    return _

>>> computation = bind( read('a'),  lambda a:
                  bind( read('b'),  lambda b:
                        unit( a+b   ))
<a function of env>
>>> computation({'a': 7, 'b': 3})
10
```

reader monad

```
def unit(v): return lambda env: v
def bind(mv, mf): …

def read(key):
    def _(env):
        return env[key]
    return _

>>> computation = bind( read('a'),  lambda a:
                  bind( read('b'),  lambda b:
                        unit( a+b   ))
<a function of env>
>>> computation({'a': 42, 'b': 1})
43
```

environment = reader + writer

```
def unit(v): …
def bind(mv, mf): …

def read(key): lambda env: (env[key], env)
def write(key, val): lambda env: … (None, newEnv)

>>> write('a', 2)
<a function of env>
>>> myenv = {'a': 999, 'b': 999}
>>> write('a', 2)(myenv)
(None, {'a': 2, 'b': 999})
>>> read('a')(myenv)
(999, {'a': 999, 'b': 999})
```

environment = reader + writer

```
>>> computation = bind( read('a'),        lambda a:
                  bind( read('b'),        lambda b:
                  bind( write('c', a+b),  lambda _:
                        unit(c+1)         )))
<a function of env>


>>> computation({'a': 7, 'b': 3})
(11, {'a': 7, 'b': 3, 'c': 10})
```

interpreter = environment + error

```
def read(key): lambda env: ...         # can fail
def write(key, val): lambda env: ...  # can fail


>>> computation = bind( read('a'),         lambda a:
                  bind( read('b'),         lambda b:
                  bind( write('c', a+b),  lambda _:
                        unit(c+1)        )))


>>> computation({'a': 7, 'b': 3})
( (11, {'a': 7, 'b': 3, 'c': 10}), None)


>>> computation({'b': 3})
( (None, {'b': 3}), "unbound symbol 'a'")
```

interpreter = environment + error

```
lis.py> 1
((1, {}), None)
lis.py> x
((None, {}), 'referenced unbound symbol x')
lis.py> (define x 1)
((None, {'x': 1}), None)
lis.py> x
((1, {'x': 1}), None)
lis.py> (+ x 2)
((3, {'x': 1}), None)
lis.py> (set! x (+ x 2))
((None, {'x': 5}), None)
lis.py> x
((5, {'x': 5}), None)
```

interpreter = environment + error

```
lis.py> +
((<function <lambda> at 0x10047f320>,
{
'+': <function <lambda> at 0x10047f320>
'*': <function <lambda> at 0x10047f398>,
'dumpenv': <function <lambda> at 0x10047ecf8>,
'assert': <function <lambda> at 0x10047ec80>,
'symbol?': <function <lambda> at 0x10047eed8>,
'eq?': <function <lambda> at 0x10047f1b8>,
'car': <function <lambda> at 0x10047f2a8>,
... snip ...
}), None)
```

interpreter = monad heaven

- identity -> values
- reader -> read-only globals (builtins)
- environment -> def, set!, lambda (lexical scope)
- error -> assert, throw
- continuation -> call/cc, try/catch/finally

"monads are how you implement special forms"

parser = environment + maybe

- env where you are in the input, not a map
- \`nil\` indicates failed parse branch, discard current env, restore env to last successful parse point and try another branch

So who cares?

- Monads are a tool to manage complexity
- Write code with maximally separated concerns
- trust that you won't break existing code, allow you to maintain with minimal changes
- write a language to talk about business state transitions, write your business logic in that language