16 September 2013

Monads - function composition on steroids

Once you start learning functional programming (whatever language you choose) sooner or later you will hit on Monad.

Your initial understanding of Monad concept, after reading a couple of tutorials (many of them are available in the web), might not be very good and that's because of The Curse of the Monad:

> In addition to it begin useful, it is also cursed and the curse of the monad is that once you get the epiphany, once you understand - "oh that's what it is" - you lose the ability to explain it to anybody.
> -- Douglas Crockford

Although the curse might be real, I'm writing here yet another monad tutorial because, what I believe, the best way to understand monad is to write tutorial about it :-)

Monad - introduction

The concept originates from category theory. In functional programming:

monad as generic concept describes how to build chains/pipelines of operations while concrete monad type defines what it actually means to chain/compose operations.

Thus we can extract two concepts here:

- **Monad** - provides the abstract interface for monadic operations
- **Monadic type** - a particular class that implements monadic operations

Thinking of Monad as interface and monadic type as its implementation might be a good initial approach, but we need to keep in mind that we are talking about very generic concepts here. Scala (we will use Scala to implement some examples) does not even provide abstract representation of Monad (there is no Monad trait). Despite this, Scala supports monadic types (more on this later on), and the only requirement for any class to be monadic type is to implement monadic operations. Haskell, on the other hand, being a purely functional language provides abstract representation of Monad (in form of class constructor). I mention Haskell because it was the first language that popularized use of monads.

Generally speaking monads are used to make things composable. Many different things (such as computations or data containers) can be expressed as monadic types. Because of that universal nature of monads, many programming functional languages provide special syntax (Haskell, Scala) or functions (Clojure) to make composition of monadic types easy to write in the code.

Before we get to monadic types, we will first see how function composition and application work and then we will see how to use those concepts to build composable structures (monadic types). We will use Haskell notation to define functions as it is very concise.

Function application

Lets define function `f` that returns its argument multiplied by 2:

```
f :: Int -> Int // f takes int, returns int

f = \x -> 2 * x // f is constructed using lambda expression (anonymous function), thus \x -> ...
```

Then:

```
f 2 --> has value 4
```

Now, calling a function with an argument can be generalized as a new function that calls given function (passed as first argument) with given argument (passed as second argument), or in other words, applies given function to given argument. This new function can be expressed as function application operator (`$` - in Haskell) :

```
($) :: (a -> b) -> (a -> b)
// $ takes function, returns function, this is the same as ((a -> b), a) -> b, instead of passing all arguments at once, we can pass first argument to function, receive partially applied function, and pass second argument to it. This is called currying (http://en.wikipedia.org/wiki/Currying).

f $ 2  --> has value 4
// $ is an infix operator, so it must be putted between its arguments

($) f 2  --> has value 4
// Putting parenthesis around an infix operator converts it into a prefix operator
```

About me

Paweł Kaczor - software developer, passionate about functional programming, the Scala programming language and the DDD/CQRS/ES architecture.

Add me on Google+

Paweł Kaczor

In 19 circles

If we define *reverse apply* operator:

`(>$>) :: a -> (a -> b) -> b`

we can express function application as following:

`2 >$> f`

Notice that unix shell's pipe (|) operator works this way - you produce some data and then apply a program to it to transform it in some way. The arrows in `>$>` indicate the direction of data flow (result from previous operation is forwarded to next operation).

Function composition

Let's say we have two functions `f` and `g` and a value `x` with the following types:

```
x :: a
f :: a -> b
g :: b -> c
```

for some types a, b, and c.

We can create new function `h :: a -> c` that combines f and g.

`h = g . f = \x -> g (f x)`

So, composing two functions produces a new function that, when called with a parameter `x` is the equivalent of calling `f` with the parameter `x` and then calling the `g` with that result.

Note that this will only work if the types of `f` and `g` are compatible, i.e. if the type of the result of `f` is also the type of the input of `g`.

Example usage:

```
f = \x -> x + 2
g = \x -> x * 3
h = g . f
h 2 --> has value 12
```

Composition in terms of application

We can easily discovery that `h` can be defined as **application of `g` to result of `f`**.
To better visualize data flow we need another operator that is reversed version of function composition operator `.`:

`h = f >.> g`

So, the key here is to understand that in order to combine two functions we need to provide a new function that calls second function with result of first function.
As it will turn up, `>.>` function is the core ingredient of monadic type.

Monadic type

So far we have only considered functions as units of composition. But what we want to achieve is composable structure / class.
Lets introduce classes to our functions and start thinking how we can compose them applying the rules of function composition.
To implement examples, we will use **Scala** - hybrid functional-OO language.

Lets introduce abstract generic class `Monad[A]` and simple class `IntWrapper` that will extend `Monad` and wrap integer value:

```
trait Monad[A] {}
case class IntWrapper(value: Int) extends Monad[Int] {}
```

Now, instead of function `Int -> Int` lets define function `Int -> Monad[Int]`

```
f :: Int -> Monad[Int]
f = \x -> IntWrapper(x + 2)

g :: Int -> Monad[Int]
g = \x -> IntWrapper(x * 3)
```

So, how we can compose those functions?

`h = f >.> g // won't work`

We can't use `>.>` operator because the types do not match. It is not possible to apply `Int ->`

`Monad[Int]` to `Monad[Int]`. We need `Int`, not `Monad[Int]`
The solution would be to create a different composition operator/function that would extract `Int` from `Monad[Int]` before performing application but that would require `Monad` to provide some kind of extract method. That doesn't sound like a good idea (and in fact would break the whole idea of Monad).

As a great OO developer you have probably already come up with the solution. `Monad` itself should provide this function as a method! That way, the class itself will define what does it mean to compose it with another monad of the same type! This method is one of the two fundamental monadic operations, it is usually called **bind**, in Haskell it is **>>=**, in Scala it is **flatMap**. Lets stay with a Haskell notation for a moment.

Notice that `>>=` is very similar to `>.>` (regular function application operator), except that it works for "monadic functions" - functions returning monadic type.

So now, assuming our `IntWrapper` implements `>>=` we can define `h` function that combines `f` and `g`:

`h = f >>= g`

or, keeping in mind that `h` must be called with a parameter that it will be passed to `f`, we can define `h` as following:

`h = \x -> f x >>= g`

Conceptually `>>=` is still a function application operator (takes a result of one function and applies another function to it):

```
(Int -> Monad[Int]) >>= (Int -> Monad[Int])
```

but in fact `>>=` is a method of monadic type with the following signature:

```
>>= :: (Int -> Monad[Int]) -> Monad[Int]
```

The method `>>=` must apply given function `Int -> Monad[Int]` to some `Int` (it must call given function with some argument `Int`), but what exactly will be the value of the argument is decided by the monadic type itself (our `IntWrapper` may decide just to pass a value member to the function).

With the implementation of `>>=` monadic type describes the meaning of composition. It defines what does it mean to compose it with another monad of the same type

We can also say, that by implementing `>>=` monadic type defines how to apply given function to itself.

Lets see how we can compose `IntWrapper`s, assuming `IntWrapper` defines composition as just passing an `Int` value forward:

```
case class IntWrapper(value: Int) extends Monad[Int] {
  override def >>=(f: Int => Monad[Int]): Monad[Int] = {
    f(value)
  }
}
def h(a: Int) = {
  IntWrapper(a + 2) >>= (b => IntWrapper(b * 3))
}
h(2) // <-- has value IntWrapper(12)
```

OK, this is not very usable application of monad as nothing is happening during composition (in the background).
In next, still simple (or even stupid simple, but more funny) example we will define one more monadic operation that will let us take use of syntactic sugar provided by Scala to compose monads more easily.

Uncertainty principle

In quantum mechanics, there is a rule saying that there are pairs of physical properties of particle (observables) known as complementary variables, that can not be measured simultaneously with arbitrarily precision. Complementary variables are for example position and momentum (of a particle).

We can encapsulate uncertainty of measurement inside `Observable` monadic type by implementing `>>=` appropriately ensuring that any calculation that takes any number of `Observable`s can only be performed as monadic operation, meaning `Observable`s must be composed in order to unveil their values that are required to calculate the result (another `Observable`).

Putting it differently, `Observable` will not allow accessing its value directly, but only when used in special context (we can name it as laboratory or measurement) which can be easily created using Scala's *for comprehension* - syntax to simplify monadic composition.

The essential part will be of course `>>=` method of `Observable` class. What we can do is to create a new value as a sum of internal (real) value property of Observable and some random value (between 0 and 1) whenever composition occurs (to simulate the distortion of one of the two

observables that are measured) and forward this new value in composition chain:

```
case class Observable[A <: Number](value: A) extends Monad[Number] {
  // remember, flatMap is what Scala expects as monadic apply (aka bind) operator, it is >>= in
Haskell
  override def flatMap[B](f: Number => Monad[B]): Monad[B] = {
    f(Math.random() + value.doubleValue())
}
```

Now we can define a method that performs measurement and calculates some value in a result:

```
def combinationOfPositionAndMomentum = for /*start measuring*/ {
  position <- Observable(4)
  momentum <- Observable(6)
} yield /*calculate result*/ {
  BigDecimal.valueOf(
    if (position.doubleValue() > 4.5) {
      momentum.doubleValue() + position.doubleValue()
    } else {
      momentum.doubleValue() - position.doubleValue()
    }
  )
}

combinationOfPositionAndMomentum // <-- has value Observable(10.95)
combinationOfPositionAndMomentum // <-- has value Observable(1.83)
```

After executing two experiments and checking the results (10.95 and 1.83) we can conclude that uncertainty principle is a fact ;)
Well, what we can say for sure is that `flatMap` method has been called what indicates that composition took place.

As you can see there is no invocation of `flatMap` method on first `Observable` in the code. This method call is generated automatically by the compiler. The function which is passed to `flatMap` method of first `Observable` is created by the compiler and basically what it does it takes the expression inside `yield` block and creates monadic type from it (wraps it into monadic type). Complete code that is generated looks like this:

```
Observable(4) flatMap
  (position => Observable(6).map(
   momentum => //content of yield (result calculation)
  )
```

As you can see monadic type `M[A]` is expected to implement method `map :: A -> B -> M[B]` (second monadic operation next to `bind`) that is able to create `M[B]` using given function `A -> B` assuming monadic type provides "constructor" applicable to `B`.

In our example:

- type of input observables is `Observable[Integer]`
- type of expression inside yield block is `BigDecimal`
- type of the whole expression (`for` in Scala is an expression, it returns value) is `Observable[BigDecimal]`

therefore, invocation of `map` on the last observable in a block is necessary to do the "conversion" from `BigDecimal` to `Observable[BigDecimal]`.

Now, lets do the final experiment:

```
val singleMeasurement = for {
  v1 <- Observable(4)
} yield {
   v1 + 4
}
singleMeasurement // <--- has value Observable(8)
```

Obviously this time `flatMap` was not invoked, just `map`. So, as expected, when we do some calculations with only one `Observable`, distortion of measurement does not occur.

You can check the complete source code here.

Lets see now some examples of monads that are available in Scala.

Some well known monads

Option

`Option[A]` - represents a value that is either a single value of type A, or no value at all. **This is a hammer for nulls and helps avoiding *NullPointerException***. When dealing with several optional values (instances of `Option`), we avoid checking for null (using nested if statements) by simply using monadic composition.

```
val map: Map = ...
```

```
for {
  value1 <- map.get("key1")
  value2 <- map.get("key2")
} yield {
 value1 + value2
}
// result is Option[Nothing] if either value1 or value2 is null
// As soon as null is detected in the chain of function calls, Option[Nothing] is propagated to the end of the chain.
```

List

`List[A]` - you might be surprised, but list is a monad too. Thats why you can compose lists easily:

```
for {
  v1 <- List(1, 3, 5)
  v2 <- List("2", "4", "6")
  if v1 < v2.toInt
} yield {
 v1 * 10 + v2.toInt
}
// result is List(12, 14, 16, 34, 36, 56)
```

Future

`Future[A]` - represents a value of type A being a result (initially unknown) of some operation. Helps chaining operations that are executed asynchronously. See: Future is a Monad

```
def slowCalcFuture: Future[Int] = ...
val future1 = slowCalcFuture
val future2 = slowCalcFuture
def combined: Future[Int] = for {
  r1 <- future1
  r2 <- future2
} yield r1 + r2
```

Either (from Scala core) or Validation (from scalaz)

`Either[A, B]` - represents value that is either A or B (typically error or result of computation). Helps hiding the boilerplate "try/catch logic".
`Validation` (from scalaz) allows to accumulate errors (`Validation` is actually not a monad but applicative functor).

```
def costToEnter(p : Person) : Validation[String, Double] = {
    for {
      a <- checkAge(p)
      b <- checkClothes(a)
      c <- checkSobriety(b)
    } yield (if (c.gender == Gender.Female) 0D else 5D)
  }
//example source code found here: https://gist.github.com/oxbowlakes/970717
```

Other examples

- Towards an immutable domain model - monads - interesting way of handling validation and operation chaining in event sourced aggregates using monadic type (Behavior)

Composing monads

It may sound strange but monads generally do not compose with each other. But of course, there is always a workaround: monad transformers. We will not go so far.. I will end up here.

Posted by Paweł Kaczor

Labels: Functional programming, Scala

3 comments:

Yirie 15 July 2015 at 11:02

Thanks for the tutorial. It's the best I have found. I completely skipped the Uncertainty principle. It was too far fetched for me. What do you mean by "monads generally do not compose with each other" when the title is "Monads - function composition on steroids". It kind of refutes the title.

Reply

Paweł Kaczor 15 July 2015 at 13:50

Thanks Yirie for your feedback. Monads do not compose because you can not compose

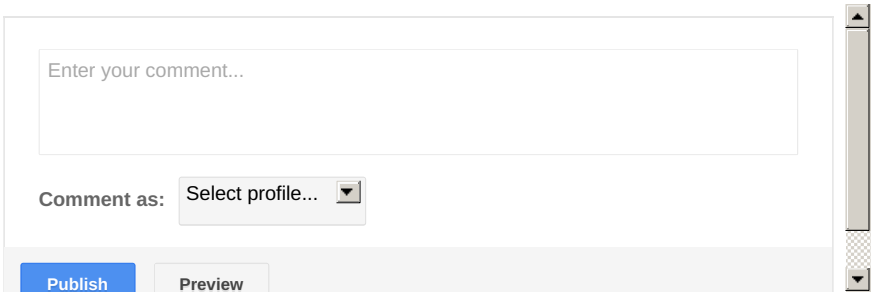
Monad[A] with Monad[B]. As mentioned in the article a workaround is to use Monad Transformers (http://book.realworldhaskell.org/read/monad-transformers.html).

Reply

Teck Hooi Lim 25 November 2015 at 11:39

How do I re-implement the Observable example using Scala Numeric typeclass of Java's Number?

Reply



Newer Post Home Older Post

Subscribe to: Post Comments (Atom)