9 — An Algebraic View On More Monads

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2020

Lehrstuhl für Informatik VI, Uni Würzburg

Recap: Algebras

Algebra

An algebra is one or more sets of values (types), together with functions operating on values of those sets and a collections of axioms.

Always thinking about a monad at implementation level is cumbersome. If we don't want to write a new monad, knowing about its algebra should be enough. We will look at some new monads in this lecture, without going into detail on how they are implemented.

The Reader Monad

The Reader Monad

Let's imagine we have a very simple web server.

On every request, the framework gathers for us all the useful information regarding that request:

```
final case class Request(
  user:   Option[String],
  locale: String,
  route:  String,
  params: Map[String, List[String]],
  now:    LocalDateTime,
  )
```

All we have to do is implement the trait `Server`:

```
trait Server:
  def serve(r: Request): String
```

The Reader Monad

Let's look at a very simple implementation:

```
object Server1 extends Server:
  def serve(r: Request): String = r.route match
    case "/hello" => sayHello
    case "/bye"   => sayBye

  def sayHello = "Hello!"
  def sayBye   = "Bye!"
```

This is cool, but we want more. We want to customize the messages depending on time and user.

The Reader Monad

```
object Server2 extends Server:
  def serve(r: Request): String = r.route match
    case "/hello" => sayHello(r)
    case "/bye"   => sayBye(r)

  def sayHello(r: Request): String =
    val user = formatUser(r)
    val time = formatTime(r)
    s"Hello $user, now is $time"

  def sayBye(r: Request): String =
    val user = formatUser(r)
    val time = formatTime(r)
    s"Goodbye $user, now is $time"

  def formatUser(r: Request): String = r.user.getOrElse("anonymous")
  def formatTime(r: Request): String = r.now.toString
```

The Reader Monad

Now this works. We just pass the request down to every function which uses it. But can we do better than manually passing the request around?

We could:

- Store the request in a global variable.
  No way. Needs side effects, hides depedency.
- Store the request in a thread local variable.
  Same problem, doesn't work well with fibers and the like.
- Use a dependency injection framework.
  Depending on the DI framework, has the same and/or other drawbacks.

It would be nice, if we could abstract over asking for some piece of data. **Reader Monad** to the rescue!

The Reader Monad

Let's look at a new monad, the reader monad. Its type is `Reader[-R, A]` and it represents a computation which needs an `R` to produce an `A`.

We will look at it purely in terms of its algebra. There is only one new operation:

```
def ask[R]: Reader[R, R]
```

This gives us a new `Reader` that takes an `R` and produces an `R`.

The Reader Monad

And since `Reader` is a monad, there are of course the monad operations:

```
def flatMap[A,B](fa: Reader[R,A])(f: A => Reader[R,B]): Reader[R,B]
def pure[A](a: A): Reader[R,A]
def map2[A,B,C](fa: Reader[R,A], fb: Reader[R,B])(f: (A,B) => C): Reader[R,C]
def map[A,B](fa: Reader[R,A])(f: A => B): Reader[R,B]
```

Note that they all operate on the second type parameter of the `Reader` only, i.e. the produced value. The input always stays the same.

With only that in mind, you should be able to write:

```
def formatUser: Reader[Request, String] =
```

which should return the user name if there is one and `"anonymous"` otherwise. Here's Reader's only non-monad method and the Request class again:

```
def ask[R]: Reader[R, R]
```

```
final case class Request(
  user:   Option[String],
  locale: String,
  route:  String,
  params: Map[String, List[String]],
  now:    LocalDateTime,
  )
```

The Reader Monad

Solution:

```
def formatUser: Reader[Request, String] =
  for
    request <- ask[Request]
  yield request.user.getOrElse("anonymous")
```

After that, it should be pretty easy to write:

```
def formatTime: Reader[Request, String] =
```

which should just invoke `toString` on the date and return that. Request being:

```
final case class Request(
  user:   Option[String],
  locale: String,
  route:  String,
  params: Map[String, List[String]],
  now:    LocalDateTime,
  )
```

The Reader Monad

Solution:

```
def formatTime: Reader[Request, String] =
  for
    request <- ask[Request]
  yield request.now.toString
```

The real power comes when combining those monadic operations to form bigger ones:

```
def sayBye: Reader[Request, String] =
```

which should just format the user, then the time, and then build a string like `s"Goodbye $user, now is $time"`. Request being:

```
final case class Request(
  user:   Option[String],
  locale: String,
  route:  String,
  params: Map[String, List[String]],
  now:    LocalDateTime,
  )
```

Solution:

```
def sayBye: Reader[Request, String] =
  for
    user <- formatUser
    time <- formatTime
  yield s"Goodbye $user, now is $time"
```

Note that we didn't need to mention the request at all, nor did we `ask` for it.

The Reader Monad

This allows us to express the whole server this way:

```
object Server3:
  def serve: Reader[Request, String] = ask[Request].flatMap(r => r.route match {
    case "/hello" => sayHello
    case "/bye"   => sayBye
  })

  def sayHello: Reader[Request, String] = for
    user <- formatUser
    time <- formatTime
  yield s"Hello $user, now is $time"

  def sayBye: Reader[Request, String] = for
    user <- formatUser
    time <- formatTime
  yield s"Goodbye $user, now is $time"

  def formatUser = ask[Request].map(_.user.getOrElse("anonymous"))
  def formatTime = ask[Request].map(_.now.toString)
```

The Reader Monad

As a side note, let's see what the reader monad actually is:

```
case class Reader[-R, A](run: R => A)
```

So our Reader actually encapsulates a simple function. All monadic operations like `map` and `flatMap` are just function composition and don't deal with a value of type `R` at all but add transformations to the result of the `run` function.

To actually produce a value, we run the Reader's function at the end, just do `readerValue.run(value)` and it will produce the result from the input value.

The Reader Monad

The reader monad allows us to do dependency injection in a type safe way, removes most of the boilerplate and hides no dependencies.

We can always ask for the value we need and never need to explicitly pass it down to the functions we use.

The Writer Monad

The Writer Monad

Let's start with a mathematical problem. Given the function

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

it is conjectured that given any positive integer, taking that integer and successively applying the function to it will always reach 1 after a finite number of steps.

This is called the Collatz conjecture.

The Writer Monad

We can define the collatz depth of a number to be the number of successive applications of the function needed to yield 1.

Now we want to find the smallest number which has at least a given collatz depth. We can write this mostly by wrapping the formula from before in a recursion:

```
def collatzDepth(n: Int): Int =
  if n == 1 then
    0
  else if n % 2 == 0 then
    collatzDepth(n/2) + 1
  else
    collatzDepth(n * 3 + 1) + 1

def collatzSearch(start: Int, limit: Int): Int =
  if collatzDepth(start) < limit then collatzSearch(start + 1, limit)
  else start
```

The Writer Monad

This is simple. But what happens when we do not only want to know the number, but also the calculations which have been needed to arrive at the solution?

We might want something like this.

```
testing 1
got 1, doing nothing
depth was 0
testing 2
got 2, halving
got 1, doing nothing
depth was 1
testing 3
got 3, tripling plus one
got 10, halving
got 5, tripling plus one
got 16, halving
got 8, halving
got 4, halving
got 2, halfing
got 1, doing nothing
depth was 7
returning 3
```

The Writer Monad

We can just go about and return a tuple of a log and the value instead of just the value. The inner function would look like this:

```
def collatzDepth(n: Int): (List[String], Int) =
  if n == 1 then
    (List("got 1, doing nothing"), 0)
  else if n % 2 == 0 then
    val (way, depth) = collatzDepth(n/2)
    (s"got $n, halfing" :: way, depth + 1)
  else
    val (way, depth) = collatzDepth(n * 3 + 1)
    (s"got $n, tripling plus one" :: way, depth + 1)
```

The Writer Monad

The outer function would look like this:

```
def collatzSearch(start: Int, limit: Int): (List[String], Int) =
  val (way, depth) = collatzDepth(start)
  if depth < limit then
    val (way2, number) = collatzSearch(start + 1, limit)
    (s"testing $start" :: way ++ (s"depth was $depth" :: way2), number)
  else
    (s"testing $start" :: way ++ List(s"depth was $depth", s"returning $start"),
    ↪  start)
```

The Writer Monad

Obviously, this is very hard to read and it is extremely error prone to get the order of the log additions right.

Another monad to the rescue. The **Writer Monad**. Again, we look at it purely in terms of its algebra.

The Writer Monad

When looking at `Writer`, we are looking at values of the type `Writer[L, A]` where `A`, as always, is the type of the value produced by the computation and `L` is a kind of log which can be written to while performing a computation.

Same as `Reader`, there is only one new operation, which is writing to the log:

```
def tell[L](l: L): Writer[L, Unit]
```

And, of course, there are also the monad operations:

```
def pure[A](a: A): Writer[L,A]
extension [A](fa: Writer[L,A])
  def flatMap[B](f: A => Writer[L,B]): Writer[L,B]
  def map2[B,C](fb: Writer[L,B])(f: (A,B) => C): Writer[L,C]
  def map[B](f: A => B): Writer[L,B]
```

The Writer Monad

But beware:

```
given [L: Monoid]: Monad[[a] =>> Writer[L, a]] with
```

That means, that you can only get a `Monad` instance for a `Writer` as long as the log type has a `Monoid` instance. This instance is needed to create an empty log on `pure` and to combine several logs on `flatMap`.

Armed with that, try to implement the collatz depth function:

```
def collatzDepth(n: Int): Writer[List[String], Int] =
```

You can find the code for the old version in the repo.

The Writer Monad

Solution:

```
def collatzDepth(n: Int): Writer[List[String], Int] =
  if n == 1 then
      tell(List("got 1, doing nothing")).map(_ => 0)
  else if n % 2 == 0 then
    for
      _     <- tell(List(s"got $n, halving"))
      depth <- collatzDepth(n/2)
    yield depth + 1
  else
    for
      _     <- tell(List(s"got $n, tripling plus one"))
      depth <- collatzDepth(n * 3 + 1)
    yield depth + 1
```

Now try to implement the collatz search function

```
def collatzSearch(start: Int, limit: Int): Writer[List[String], Int] =
```

Again, the code for the old version can be found in the repo.

Solution:

```
def collatzSearch(start: Int, limit: Int): Writer[List[String], Int] =
  for
    _      <- tell(List(s"testing $start"))
    depth  <- collatzDepth(start)
    _      <- tell(List(s"Depth was $depth"))
    result <- if depth < limit then collatzSearch(start + 1, limit)
              else tell(List(s"Returning $start")).map(_ => start)
  yield result
```

The Writer Monad

Now, lets take a quick look at what a `Writer` really is:

```
final case class Writer[L, A](v: (L, A))
```

It's nothing more than a wrapper around a tuple of a log together with a value. That means, to extract the value of an `Writer`, all we have to do is `writerValue.v._2`. And to get the log, all we have to do is `writerValue.v._1`.

Handling Mutable State

Mutable State

Until now, we avoided mutable state, because it breaks referential transparency. But some things are much easier to express using mutable state, for example a pseudorandom number generator (PRNG).

We will now see, how we can encapsulate mutable state in a purely functional, and therefore referentially transparent way.

Stateful Random Numbers

Let's take a simple, stateful example:

```
val r = new Random
println(r.nextDouble)
println(r.nextDouble)
```

Which gives us:

```
0.1555342604626314
0.7087659049981789
```

Which is clearly not functional. How do we proof that?

Stateful Random Numbers

Apart from seeing it directly we could write two programs:

```
val r = new Random
println(r.nextDouble == r.nextDouble)
```

```
val r = new Random
val a = r.nextDouble
println(a == a)
```

Which gives us:

```
false
```

```
true
```

So by extracting an expression into a variable, we have changed the meaning of our program. This proofs that this expression is not purely functional.

Stateful Random Numbers

Why is this bad?

```
def rollDie: Int =
  val r = new Random
  r.nextInt(6)
```

This implementation of a dice with numbers 1–6 has a bug. But it's not easily reproducible.

```
def rollDie(r: Random): Int =
  r.nextInt(6)
```

Dependency injection to the rescue! But still hard to reproduce, because you have to generate a new `Random` for every test and then throw it away again.

Stateful Random Numbers

Let's change course and try to do it without side effects.

```
trait RNG:
  def nextInt: (RNG, Int)
```

Which lets us write code like

```
val r1: RNG = ??? // a possible implementation follows later
val (r2, i1) = r1.nextInt
val (r3, i2) = r2.nextInt
val (r4, i3) = r3.nextInt
```

Instead of relying on implicit state changes, we never mutate state and explicitly pass the new state along.

Stateful Random Numbers

Before we start addressing the obvious problem that it is extremely tedious to pass this state along manually, let's create a simple **RNG** implementation[1].

```
case class Simple(seed: Long) extends RNG:
  def nextInt: (RNG, Int) =
    val newSeed = (seed * 0x5DEECE66DL + 0xBL) & 0xFFFFFFFFFFFFL
    val nextRNG = Simple(newSeed)
    val n = (newSeed >>> 16).toInt
    (nextRNG, n)
```

Important facts:

- has a seed which is its state
- doesn't mutate its seed but generates a new seed for the next RNG

[1] This is basically the approach and constants java.util.Random uses

Stateful Random Numbers

We can now implement various random generators based on the RNG trait:

```
def nonNegativeInt(rng: RNG): (RNG, Int) =
  val (r, i) = rng.nextInt
  (r, if i < 0 then -(i + 1) else i)
```

```
def double(rng: RNG): (RNG, Double) =
  val (r, i) = nonNegativeInt(rng)
  (r, i / (Int.MaxValue.toDouble + 1))
```

```
def boolean(rng: RNG): (RNG, Boolean) =
  val (r, i) = rng.nextInt
  (r, i % 2 == 0)
```

All have in common, that they receive a RNG state, and return a new RNG state together with their result.

state

We said we don't want to pass the state around all the time. Similarly to Reader and Writer, we want a structure, that handles the state keeping for us.

We will use the `State` monad, whose type is `State[S, A]`. Like `Reader`, it represents a computation, that needs some `S` to generate an `A`.

The difference is, that `Reader[R,A]` only combines computations that all use the same `R`, our `State[S,A]` also combines computations that can change the value of `S` (but not its type).

The State Monad

The operations for state are the following functions:

```
def get[S]: State[S, S]
def set[S](s: S): State[S, Unit]
```

And of course the monad functions:

```
def pure[A](a: A): State[S,A]
extension [A](fa: State[S,A])
  def flatMap[B](f: A => State[S,B]): State[S,B]
  def map2[B,C](fb: State[S,B])(f: (A,B) => C): State[S,C]
  def map[B](f: A => B): State[S,B]
```

Note that again, like with Reader, none of the functions are allowed to change the *type* of the state, it stays `S`. But we can change the value using `set(newState)`.

Wrapping the RNG

Let's wrap RNG in a State monad:

```
def randomInt: State[RNG, Int] =
  for
    rng       <- get[RNG]
    (rng2, i)  = rng.nextInt
    _         <- set(rng2)
  yield i
```

This doesn't look much better than manual passing of the state yet. But our derived functions become much simpler:

```
def nonNegativeInt: State[RNG, Int] =
  for i <- randomInt
  yield if i < 0 then -(i + 1) else i
```

Exercise: Chaining states

The power of the state monad becomes apparent, when doing several state actions on the same state in sequence.

Implement the following function, which returns a random tuple of three ints:

```
def threeInts: State[RNG, (Int, Int, Int)] =
```

Remember that State handles all passing around of the RNGs for you.

Exercise: Chaining states

Solution:

```
def threeInts: State[RNG, (Int, Int, Int)] =
  for
    a <- randomInt
    b <- randomInt
    c <- randomInt
  yield (a,b,c)
```

State: Under the hood

So, what does our `State` monad actually look like?

```
case class State[S,+A](run: S => (S, A))
```

It wraps a function that takes the current state and returns the next state and an output. Our monad combinators are doing exactly what we did when we manually threaded the state through all calls, hiding this passing for us.

Like with `Reader`, we only compose functions. To produce the results, we call `run` with the initial state in the end.