This is part 2 of a multipart blog series on a bunch of useful monads that scalaz offers that the scala standard library doesn't have.

It is assumed that you have read part 1 for this blog. You can find it here (https://functionaltechramblings.wordpress.com/2016/02/28/monads-and-scalaz-what-when-why-and-how/) if you haven't checked it out yet.

Now for the next exhilarating installment…

Writer

What

The Writer monad is effectively a wrapper around a tuple

```
(W, A)
```

Put simply, it is a way to return a value `A` along with some extra *written* information. `W` typically stands for the *written* value and `A` stands for the result.

In scalaz it is `Writer[W, A]` and is actually a type alias for `WriterT[Id.Id, W, A]` but we won't concern ourselves with what that means until the blog on monad transformers.

I have found that a nice intuition for this monad is to imagine it is a log `W` attached to a value `A`. If we assume that `W` can be combined, then we can work with the value `A` like we always would have, and combine our `W` value as we go. We will see the details of how this works shortly.

Why

Much like the motivation for the state monad, we want to avoid side effects in our code. Lets consider the standard writer example of logging.

```
def add(a: Int, b: Int): Int = {
  println(s"Adding $a and $b")
  a + b
}

def multiply(a: Int, b: Int): Int = {
  println(s"Multiplying $a and $b")
  a * b
}

multiply(add(3, 4), 6)  // Also triggers two println side effects
// res0: Int = 42
```

We want to be able to capture these side effects explicitly and handle them rather than compose our program out of side effecting functions.

If we try this naively we might get the following

```scala
def add(a: Int, b: Int): (String, Int) = {
  (s"Adding $a and $b", a + b)
}

def multiply(a: Int, b: Int): (String, Int) = {
  (s"Multiplying $a and $b", a * b)
}

multiply(add(3, 4), 6)    // Does not compile

// We are forced to do the following kind of thing
val (w1, a1) = add(3, 4)
// w1: String = Adding 3 and 4
// a1: Int = 7
val (w2, a2) = multiply(a1, 6)
// w2: String = Multiplying 7 and 6
// a2: Int = 42

// Do something with w1 and w2
```

This is an improvement since our functions `add` and `multiply` no longer contain side effects, however dealing with these functions is no longer very nice. What would be great is if we had a way to accumulate those w1, w2 values whilst only worrying about working with the a1, a2 values.

In comes the writer monad.

How

Since the writer monad effectively acts as a wrapper for `(W, A)` our simple math functions fit the bill perfectly.

We can rewrite the above as

```scala
import scalaz._
import Scalaz._

def add(a: Int, b: Int): Writer[String, Int] = {
  (a + b).set(s"Adding $a and $b")
}

def multiply(a: Int, b: Int): Writer[String, Int] = {
  (a * b).set(s"Multiplying $a and $b")
}

val res0 = for {
  x <- add(3, 4)
  y <- multiply(x, 6)
} yield y
// res0: Writer[String,Int] = Writer((Adding 3 and 4Multiplying 7 an

res0.written
// res1: String = Adding 3 and 4Multiplying 7 and 6
res0.value
// res2: Int = 42
```

There is a lot going on here so lets have a look.

Firstly we have rewritten our method signatures to contain the writer monad: `Writer[String, Int]`. As we said before, lets think of this as simply being a wrapper around a tuple `(W, A)` which in this case is `(String, Int)`.

We constructed this writer using `set`. Set simply takes a value for the left side of the tuple `W` and writes it next to the value you call it on. So in the case of `add` it is creating a tuple `(s"Adding $a and $b", a + b)`.

It may not seem immediately obvious why this is better than just a straight tuple, but if we look at the for comprehension, we see something quite interesting. Much like our State monad, we only have to worry about the value we care about. In this case that is the `A` value. Our `W` value is dealt with as part of the flatmap/map/filter chains inside the for comprehension.

But how is `W` being “dealt with”?

To answer this question properly, we need to understand what a `Semigroup` is. Put simply, it is something that is combineable with other values of the same type.

e.g.
– An int semigroup for addition would have a combine of `_ + _`.
– An int semigroup for multiplication would have a combine of `_ * _`.

(This is a simplification to aid understanding. If you want to understand the details, refer to my previous blog (https://functionaltechramblings.wordpress.com/2015/08/21/some-clarity-on-the-useful-bits-of-category-theory-in-relation-to-fp/)).

When we have a combinable value for `W`, we can safely let our flatmaps combine them. So if our `W` type was `Int` and we provided an implicit evidence for the `Semigroup[Int]` implemented as the addition described above, our `W` values would simply add together. If we instead provided the implicit envidence for `Semigroup[Int]` as multiplication, we would get `W` as the product of the logs.

```scala
import scalaz._
import Scalaz._

implicit def intAdditionEv = new Semigroup[Int] {
  override def append(f1: Int, f2: => Int): Int = f1 + f2
}

val v1: Writer[Int, String] = "value 1".set(5)
val v2: Writer[Int, String] = "value 2".set(6)

v1.flatMap(x =>
  v2.map(y => x + y)
)(implicitly[Bind[Id.Id]], intAdditionEv)
// res0: Writer[Int, String] = (11,value 1value 2)

// This is the same as the following for comprehension (since scalaz
// provides the addition semigroup.
for {
  x <- v1
  y <- v2
} yield x + y
// res1: Writer[Int, String] = (11,value 1value 2)

// If we want to explicitly force it to use our new multiplication se
// we need to directly provide it

implicit def intMultiplicationEv = new Semigroup[Int] {
  override def append(f1: Int, f2: => Int): Int = f1 * f2
}

v1.flatMap(x =>
  v2.map(y => x + y)
)(implicitly[Bind[Id.Id]], intMultiplicationEv)
// res2: Writer[Int, String] = (30,value 1value 2)
```

Finally we can `run` our monad just like the state and it will return our `(W, A)` tuple. We can alternatively access the `written` value or the `value` by calling `written` and `value` respectively.

When

Even though it is essentially a restricted version of the state monad, it should be preferred when it is powerful enough to do the job at hand. In general you should use the least general abstraction necessary and this fits a lot of problems quite nicely.

The most common usage of the Writer monad that I am familiar with is for logging as we showed above. There are more sophisticated ways of doing this than simply returning a `String` (or more commonly `List[String]` or `Vector[String]`) such as treelog (https://github.com/lancewalton/treelog).

In our final example, we also saw that it can be used with any type so long as we know how we want to combine the values. We could for example update our wallet as we spend money with its new total `Writer[Total, Unit]`.

We can use it in general to catch side effects that can be combined.

Much like any other abstractions, it is simply another tool to get the job done. Becoming familiar with it will provide a good intuition for when a problem would be better solved with the Writer monad.

Stay tuned for part 3 on the Reader monad.