Addendum to ‘Monads do not Compose’

a temporary addendum to Part 1 of ‘Monads do not Compose’
illustrating an issue raised by Sergei Winitzki


@philip_schwarz

https://www.slideshare.net/pjschwarz/monads-do-not-compose

Sergei Winitzki

sergei-winitzki-11a6431

It is a mistake to think that a traversable monad can be composed with another monad.

It is true that, given `Traversable`, you can implement the monad's methods (pure and flatMap) for the composition with another monad (as in your slides 21 to 26), but this is a deceptive appearance.

The laws of the `Traversable` typeclass are far insufficient to guarantee the laws of the resulting composed monad. The only traversable monads that work correctly are Option, Either, and Writer.

It is true that you can implement the type signature of the `swap` function for any `Traversable` monad. However, the `swap` function for monads needs to satisfy very different and stronger laws than the `sequence` function from the `Traversable` type class.

I'll have to look at the "Book of Monads"; but, if my memory serves, the FPiS book does not derive any of these laws.

@philip_schwarz

Thank you very much for taking the time to raise this concern.

As seen in Part 1, the **Book of Monads** says “we definitely need a **swap** function if we want to combine two **monads**, **but this is not enough**. The reason is that a well-typed implementation may lead to a combined **monad** that violates one of the **monad laws**. The list **monad** is a well-known example of this.”

So it looks like similar concerns apply when **Traversable** is used to provide the **swap** function. In **FPiS** there is no example of using **composeM** to automatically wrap a **Traversable Monad** in any other **Monad**, so in Part 1 I had a go and provided an example by composing the **Option Monad** with the **Traversable List Monad**. I need to check if the resulting **Monad** satisfies the **Monad laws**.

In the meantime, see the next two slides for an example of the invalid composition of a **Monad** with a **Traversable Monad**. I compose the **List Traversable Monad** with itself and while in some cases the resulting **Monad** satisfies the **monadic associative law**, in another case (mentioned in the **Book of Monads**), it doesn’t.

```scala
// sample A => F[B] function
val f: String => List[String] = _.split(" ").toList.map(_.capitalize)
assert(f("dog cat rabbit parrot")
       == List("Dog", "Cat", "Rabbit", "Parrot"))

// sample B => F[C] function
val g: String => List[Int] = _.toList.map(_.toInt)
assert(g("Cat") == List(67, 97, 116))

// sample C => F[D] function
val h: Int => List[Char] = _.toString.toList
assert(h(102) == List('1', '0', '2'))

// sample A value
val a = "dog cat rabbit parrot"
// sample B value
val b = "Cat"
// sample C value
val c = 67
// sample D value
val d = '6'

// expected value of applying to "dog cat rabbit parrot"
// the kleisli composition of f, g and h
val expected =  List(
  /* Dog    */ '6', '8', '1', '1', '1', '1', '0', '3',
  /* Cat    */ '6', '7', '9', '7', '1', '1', '6',
  /* Rabbit */ '8', '2', '9', '7', '9', '8', '9', '8', '1', '0', '5', '1', '1', '6',
  /* Parrot */ '8', '0', '9', '7', '1', '1', '4', '1', '1', '4', '1', '1', '1', '1', '1', '6')

// monadic associative law: x.flatMap(f).flatMap(g) == x.flatMap(a => f(a).flatMap(g))
// alternative formuation using kleisli composition: compose(compose(f, g), h) == compose(f, compose(g, h))

// the traversable list monad satisfies the associative law because its kleisli composition is associative
assert( traversableListMonad.compose(traversableListMonad.compose(f,g),h)("dog cat rabbit parrot") == expected)
assert( traversableListMonad.compose(f,traversableListMonad.compose(g,h))("dog cat rabbit parrot") == expected)
```

```scala
// Kleisli composition (defined on Monad F)
def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] = {
  a => flatMap(f(a))(g)
}

// Monad composition
def composeM[G[_],H[_]](implicit G: Monad[G], H: Monad[H], T: Traverse[H]):
Monad[({type f[x] = G[H[x]]})#f] = new Monad[({type f[x] = G[H[x]]})#f] {
  def unit[A](a: => A): G[H[A]] = G.unit(H.unit(a))
  override def flatMap[A,B](mna: G[H[A]])(f: A => G[H[B]]): G[H[B]] = {
    G.flatMap(mna)(na => G.map(T.traverse(na)(f))(H.join))
  }
}
```

```scala
// Let's define a traversable List Monad
val traversableListMonad = new Monad[List] with Traverse[List] {
  def unit[A](a: => A): List[A] = List(a)
  override def flatMap[A,B](ma: List[A])(f: A => List[B]): List[B] = ma flatMap f
  override def map[A,B](m: List[A])(f: A => B): List[B] = m map f
  override def join[A](mma: List[List[A]]): List[A] = mma.flatten
  override def traverse[M[_],A,B](as: List[A])
    (f: A => M[B])(implicit M: Applicative[M]): M[List[B]] = {
    as.foldRight(M.unit(List[B]()))((a, fbs) => M.map2(f(a), fbs)(_ :: _))
  }
}
```

```
// compose traversableListMonad with itself
val listListMonad = composeM(traversableListMonad,traversableListMonad,traversableListMonad)

// Now let's tweak f, g, and h to return a List of a List rather than just a List, so that they are amenable to
// composing using listListMonad's kleisli composition, whose signature is
// def compose[A,B,C](f: A => List[List[B]], g: B => List[List[C]]): A => List[List[C]]
val ff: String => List[List[String]] = s => List(f(s),f(s))
val gg: String => List[List[Int]] = s => List(g(s),g(s))
val hh: Int => List[List[Char]] = n => List(h(n))

// listListMonad appears to satisfy the associative law
assert( listListMonad.compose(listListMonad.compose(fff,ggg),hhh)("dog cat rabbit parrot") == List.fill(32)(expected))
assert( listListMonad.compose(fff,listListMonad.compose(ggg,hhh))("dog cat rabbit parrot") == List.fill(32)(expected))

// but it doesn't: here is an example (by Petr Pudlak) of a function for which kleisli composition is not associative
def v(n: Int): List[List[Int]] = n match {
  case 0 => List(List(0,1))
  case 1 => List(List(0),List(1))
}

// listListMonad's kleisli composition is not associative when we compose function v with itself
assert( listListMonad.compose(listListMonad.compose(v,v),v)(0)
        != listListMonad.compose(v,listListMonad.compose(v,v)))

assert( listListMonad.compose(listListMonad.compose(v,v),v)(0)
        == List(List(0,1,0,0,1),List(0,1,1,0,1),List(0,1,0,0),List(0,1,0,1),List(0,1,1,0),List(0,1,1,1)) )

assert( listListMonad.compose(v,listListMonad.compose(v,v))(0)
        == List(List(0,1,0,0,1),List(0,1,0,0),List(0,1,0,1),List(0,1,1,0,1),List(0,1,1,0),List(0,1,1,1)) )
```