23

Comonads

Now that we have covered monads, we can reap the benefits of duality and get comonads for free simply by reversing the arrows and working in the opposite category.

Recall that, at the most basic level, monads are about composing Kleisli arrows:

```
a -> m b
```

where `m` is a functor that is a monad. If we use the letter `w` (upside down `m`) for the comonad, we can define co-Kleisli arrows as morphism of the type:

```
w a -> b
```

The analog of the fish operator for co-Kleisli arrows is defined as:

```
(=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
```

For co-Kleisli arrows to form a category we also have to have an identity co-Kleisli arrow, which is called `extract`:

```
extract :: w a -> a
```

This is the dual of `return`. We also have to impose the laws of associativity as well as left- and right-identity. Putting it all together, we could define a comonad in Haskell as:

```
class Functor w => Comonad w where
    (=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
    extract :: w a -> a
```

In practice, we use slightly different primitives, as we'll see shortly.

The question is, what's the use for comonads in programming?

23.1 Programming with Comonads

Let's compare the monad with the comonad. A monad provides a way of putting a value in a container using `return`. It doesn't give you access to a value or values stored inside. Of course, data structures that implement monads might provide access to their contents, but that's considered a bonus. There is no common interface for extracting values from a monad. And we've seen the example of the `IO` monad that prides itself in never exposing its contents.

A comonad, on the other hand, provides the means of extracting a single value from it. It does not give the means to insert values. So if you want to think of a comonad as a container, it always comes pre-filled with contents, and it lets you peek at it.

Just as a Kleisli arrow takes a value and produces some embellished result — it embellishes it with context — a co-Kleisli arrow takes a value together with a whole context and produces a result. It's an embodiment of *contextual computation*.

23.2 The Product Comonad

Remember the reader monad? We introduced it to tackle the problem of implementing computations that need access to some read-only environment `e`. Such computations can be represented as pure functions of the form:

```
(a, e) -> b
```

We used currying to turn them into Kleisli arrows:

```
a -> (e -> b)
```

But notice that these functions already have the form of co-Kleisli arrows. Let's massage their arguments into the more convenient functor form:

```
data Product e a = Prod e a deriving Functor
```

We can easily define the composition operator by making the same environment available to the arrows that we are composing:

```
(=>=) :: (Product e a -> b) -> (Product e b -> c) -> (Product e a -> c)
f =>= g = \(Prod e a) -> let b = f (Prod e a)
                             c = g (Prod e b)
```

```
                    in c
```

The implementation of `extract` simply ignores the environment:

```
extract (Prod e a) = a
```

Not surprisingly, the product comonad can be used to perform exactly the same computations as the reader monad. In a way, the comonadic implementation of the environment is more natural — it follows the spirit of "computation in context." On the other hand, monads come with the convenient syntactic sugar of the `do` notation.

The connection between the reader monad and the product comonad goes deeper, having to do with the fact that the reader functor is the right adjoint of the product functor. In general, though, comonads cover different notions of computation than monads. We'll see more examples later.

It's easy to generalize the `Product` comonad to arbitrary product types including tuples and records.

23.3 Dissecting the Composition

Continuing the process of dualization, we could go ahead and dualize monadic bind and join. Alternatively, we can repeat the process we used with monads, where we studied the anatomy of the fish operator. This approach seems more enlightening.

The starting point is the realization that the composition operator must produce a co-Kleisli arrow that takes `w a` and produces a `c`. The only way to produce a `c` is to apply the second function to an argument of the type `w b`:

```
(=>=) :: (w a -> b) -> (w b -> c) -> (w a -> c)
f =>= g = g ...
```

But how can we produce a value of type `w b` that could be fed to `g`? We have at our disposal the argument of type `w a` and the function `f :: w a -> b`. The solution is to define the dual of bind, which is called extend:

```
extend :: (w a -> b) -> w a -> w b
```

Using `extend` we can implement composition:

```
f =>= g = g . extend f
```

Can we next dissect `extend`? You might be tempted to say, why not just apply the function `w a -> b` to the argument `w a`, but then you quickly realize that you'd have no way of converting the resulting `b` to `w b`. Remember, the comonad provides no means of lifting values. At this point, in the analogous construction for monads, we used `fmap`. The only way we could use `fmap` here would be if we had something of the type `w (w a)` at our disposal. If we could only turn `w a` into `w (w a)`. And, conveniently, that would be exactly the dual of `join`. We call it `duplicate`:

```
duplicate :: w a -> w (w a)
```

So, just like with the definitions of the monad, we have three equivalent definitions of the comonad: using co-Kleisli arrows, `extend`, or `duplicate`. Here's the Haskell definition taken directly from `Control.Comonad` library:

```haskell
class Functor w => Comonad w where
    extract :: w a -> a
    duplicate :: w a -> w (w a)
    duplicate = extend id
    extend :: (w a -> b) -> w a -> w b
    extend f = fmap f . duplicate
```

Provided are the default implementations of **`extend`** in terms of **`duplicate`** and vice versa, so you only need to override one of them.

The intuition behind these functions is based on the idea that, in general, a comonad can be thought of as a container filled with values of type **`a`** (the product comonad was a special case of just one value). There is a notion of the "current" value, one that's easily accessible through **`extract`**. A co-Kleisli arrow performs some computation that is focused on the current value, but it has access to all the surrounding values. Think of the Conway's game of life. Each cell contains a value (usually just **`True`** or **`False`**). A comonad corresponding to the game of life would be a grid of cells focused on the "current" cell.

So what does **`duplicate`** do? It takes a comonadic container **`w a`** and produces a container of containers **`w (w a)`**. The idea is that each of these containers is focused on a different **`a`** inside **`w a`**. In the game of life, you would get a grid of grids, each cell of the outer grid containing an inner grid that's focused on a different cell.

Now look at **`extend`**. It takes a co-Kleisli arrow and a comonadic container **`w a`** filled with **`a`**s. It applies the computation to all of these **`a`**s, replacing them with **`b`**s. The result is a comonadic container filled with **`b`**s. **`extend`** does it by shifting the focus from one **`a`** to another and applying the co-Kleisli arrow to each of them in turn. In the game of life, the co-Kleisli arrow would calculate the new state of the current cell. To do that, it would look at its context — presumably its nearest neighbors.

The default implementation of **extend** illustrates this process. First we call **duplicate** to produce all possible foci and then we apply **f** to each of them.

23.4 The Stream Comonad

This process of shifting the focus from one element of the container to another is best illustrated with the example of an infinite stream. Such a stream is just like a list, except that it doesn't have the empty constructor:

```
data Stream a = Cons a (Stream a)
```

It's trivially a **Functor**:

```
instance Functor Stream where
    fmap f (Cons a as) = Cons (f a) (fmap f as)
```

The focus of a stream is its first element, so here's the implementation of **extract**:

```
extract (Cons a _) = a
```

duplicate produces a stream of streams, each focused on a different element.

```
duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

The first element is the original stream, the second element is the tail of the original stream, the third element is its tail, and so on, ad infinitum.

Here's the complete instance:

```
instance Comonad Stream where
    extract (Cons a _) = a
    duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

This is a very functional way of looking at streams. In an imperative language, we would probably start with a method **advance** that shifts the stream by one position. Here, **duplicate** produces all shifted streams in one fell swoop. Haskell's laziness makes this possible and even desirable. Of course, to make a **Stream** practical, we would also implement the analog of **advance**:

```
tail :: Stream a -> Stream a
tail (Cons a as) = as
```

but it's never part of the comonadic interface.

If you had any experience with digital signal processing, you'll see immediately that a co-Kleisli arrow for a stream is just a digital filter, and **extend** produces a filtered stream.

As a simple example, let's implement the moving average filter. Here's a function that sums **n** elements of a stream:

```
sumS :: Num a => Int -> Stream a -> a
sumS n (Cons a as) = if n <= 0 then 0 else a + sumS (n - 1) as
```

Here's the function that calculates the average of the first **n** elements of the stream:

```
average :: Fractional a => Int -> Stream a -> a
average n stm = (sumS n stm) / (fromIntegral n)
```

Partially applied **average n** is a co-Kleisli arrow, so we can **extend** it over the whole stream:

```
movingAvg :: Fractional a => Int -> Stream a -> Stream a
movingAvg n = extend (average n)
```

The result is the stream of running averages.

A stream is an example of a unidirectional, one-dimensional comonad. It can be easily made bidirectional or extended to two or more dimensions.

23.5 Comonad Categorically

Defining a comonad in category theory is a straightforward exercise in duality. As with the monad, we start with an endofunctor `T`. The two natural transformations, η and μ, that define the monad are simply reversed for the comonad:

$$\varepsilon :: T \to I$$
$$\delta :: T \to T^2$$

The components of these transformations correspond to **`extract`** and **`duplicate`**. Comonad laws are the mirror image of monad laws. No big surprise here.

Then there is the derivation of the monad from an adjunction. Duality reverses an adjunction: the left adjoint becomes the right adjoint and vice versa. And, since the composition $R \circ L$ defines a monad, $L \circ R$ must define a comonad. The counit of the adjunction:

$$\varepsilon :: L \circ R \to I$$

is indeed the same ε that we see in the definition of the comonad — or, in components, as Haskell's **`extract`**. We can also use the unit of the

adjunction:

$$\eta :: I \to R \circ L$$

to insert an $R \circ L$ in the middle of $L \circ R$ and produce $L \circ R \circ L \circ R$. Making T^2 from T defines the δ, and that completes the definition of the comonad.

We've also seen that the monad is a monoid. The dual of this statement would require the use of a comonoid, so what's a comonoid? The original definition of a monoid as a single-object category doesn't dualize to anything interesting. When you reverse the direction of all endomorphisms, you get another monoid. Recall, however, that in our approach to a monad, we used a more general definition of a monoid as an object in a monoidal category. The construction was based on two morphisms:

$$\mu :: m \otimes m \to m$$
$$\eta :: i \to m$$

The reversal of these morphisms produces a comonoid in a monoidal category:

$$\delta :: m \to m \otimes m$$
$$\varepsilon :: m \to i$$

One can write a definition of a comonoid in Haskell:

```
class Comonoid m where
    split :: m -> (m, m)
    destroy :: m -> ()
```

but it is rather trivial. Obviously `destroy` ignores its argument.

```
destroy _ = ()
```

`split` is just a pair of functions:

```
split x = (f x, g x)
```

Now consider comonoid laws that are dual to the monoid unit laws.

```
lambda . bimap destroy id . split = id
rho . bimap id destroy . split = id
```

Here, `lambda` and `rho` are the left and right unitors, respectively (see the definition of monoidal categories). Plugging in the definitions, we get:

```
lambda (bimap destroy id (split x))
= lambda (bimap destroy id (f x, g x))
= lambda ((), g x)
= g x
```

which proves that `g = id`. Similarly, the second law expands to `f = id`. In conclusion:

```
split x = (x, x)
```

which shows that in Haskell (and, in general, in the category **Set**) every object is a trivial comonoid.

Fortunately there are other more interesting monoidal categories in which to define comonoids. One of them is the category of endofunctors. And it turns out that, just like the monad is a monoid in the category of endofunctors,

> The comonad is a comonoid in the category of endofunctors.

23.6 The Store Comonad

Another important example of a comonad is the dual of the state monad. It's called the costate comonad or, alternatively, the store comonad.

We've seen before that the state monad is generated by the adjunction that defines the exponentials:

$$L\ z = z \times s$$
$$R\ a = s \Rightarrow a$$

We'll use the same adjunction to define the costate comonad. A comonad is defined by the composition $L \circ R$:

$$L\ (R\ a) = (s \Rightarrow a) \times s$$

Translating this to Haskell, we start with the adjunction between the `Product` functor on the left and the `Reader` functor or the right. Composing `Product` after `Reader` is equivalent to the following definition:

```
data Store s a = Store (s -> a) s
```

The counit of the adjunction taken at the object a is the morphism:

$$\varepsilon_a :: ((s \Rightarrow a) \times s) \to a$$

or, in Haskell notation:

```
counit (Prod (Reader f) s)) = f s
```

This becomes our `extract`:

```
extract (Store f s) = f s
```

The unit of the adjunction:

```
unit :: a -> Reader s (Product a s)
unit a = Reader (\s -> Prod a s)
```

can be rewritten as partially applied data constructor:

```
Store f :: s -> Store f s
```

We construct δ, or **duplicate**, as the horizontal composition:

$$\delta :: L \circ R \to L \circ R \circ L \circ R$$
$$\delta = L \circ \eta \circ R$$

We have to sneak η through the leftmost L, which is the **Product** functor. It means acting with η, or **Store f**, on the left component of the pair (that's what **fmap** for **Product** would do). We get:

```
duplicate (Store f s) = Store (Store f) s
```

(Remember that, in the formula for δ, L and R stand for identity natural transformations whose components are identity morphisms.)

Here's the complete definition of the **Store** comonad:

```
instance Comonad (Store s) where
    extract (Store f s) = f s
    duplicate (Store f s) = Store (Store f) s
```

You may think of the `Reader` part of `Store` as a generalized container of `a`s that are keyed using elements of the type `s`. For instance, if `s` is `Int`, `Reader Int a` is an infinite bidirectional stream of `a`s. `Store` pairs this container with a value of the key type. For instance, `Reader Int a` is paired with an `Int`. In this case, `extract` uses this integer to index into the infinite stream. You may think of the second component of `Store` as the current position.

Continuing with this example, `duplicate` creates a new infinite stream indexed by an `Int`. This stream contains streams as its elements. In particular, at the current position, it contains the original stream. But if you use some other `Int` (positive or negative) as the key, you'd obtain a shifted stream positioned at that new index.

In general, you can convince yourself that when `extract` acts on the `duplicate`d `Store` it produces the original `Store` (in fact, the identity law for the comonad states that `extract . duplicate = id`).

The `Store` comonad plays an important role as the theoretical basis for the `Lens` library. Conceptually, the `Store s a` comonad encapsulates the idea of "focusing" (like a lens) on a particular substructure of the data type `a` using the type `s` as an index. In particular, a function of the type:

```
a -> Store s a
```

is equivalent to a pair of functions:

```
set :: a -> s -> a
get :: a -> s
```

If `a` is a product type, `set` could be implemented as setting the field of type `s` inside of `a` while returning the modified version of `a`. Similarly,

`get` could be implemented to read the value of the `s` field from `a`. We'll explore these ideas more in the next section.

23.7 Challenges

1. Implement the Conway's Game of Life using the `Store` comonad. Hint: What type do you pick for `s`?