WORK LOG OF JUNE 19

DANIEL R. BARRERO R.

1

The comonad defnition as a typeclass is

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

And in Milewski's tic-tac-toe, *the evaluator* will be a comonad insntance, apparently. This evaluator defines a tree-to-vector behavior:

```
newtype W f a = W {runW :: forall n. f n -> Vec n a}
```

The following typings follow:

The type of `runW` is

```
runW :: W f a -> forall (n :: Nat). f n -> Vec n a
```

And that of `W` is

```
W :: (forall (n :: Nat). f n -> Vec n a) -> W f a
```

Also, `ghci` gives the following answer when asked for the kind of `W`:

```
:k W
> (Nat -> *) -> * -> *
```

Since a `Comonad` is also a `Functor`, we must first make a functor out of `W f`. For this, we first make `Vec n` into a functor via

```
instance Functor (Vec n) where
  fmap f VNil = VNil
  fmap f (VCons a vs) = VCons (f a) (fmap f vs)
```

And then `W f` with

```
instance Functor (W f) where
  fmap g (W k) =
    W (
       \ f_n -> fmap g (k f_n)
     )
```

Now, to obtain a `Comonad` instance, we must define the `extract` and `duplicate` maps. The first one is

```
extract :: W f a -> a
extract (W k) = case k ident of VCons a0 VNil -> a0
```

Given the functor he has defined, as well as the `extract` function above, one would expect the type of his `duplicate` to be

```
duplicate' :: W f a -> W f (W f a)
```

Date: June 20, 2025.

A naive function that typechecks is

```
duplicate' :: W f a -> W f (W f a)
duplicate' (W k) =
  W (
     \ t_n -> replycate (W k) (k t_n)
    )
```

Where the `replycate` function takes a value `b0` and a vector of length n with type-`a` coefficients and produces a vector of the same length with n copies of the value `b0`:

```
replycate :: b -> Vec n a -> Vec n b
replycate b0 VNil = VNil
replycate b0 (VCons a0 as) = VCons b0 (replycate b0 as)
```

This suffices to make `W f` an instance of the `Comonad` typeclass:

```
instance Operad f => Comonad (W f) where
extract = extract'
duplicate = duplicate'
```