Gathered notes from:

- Haskell Programming from First Principles [1]

1 Monoid

todo

2 Functor

todo

3 Applicative

todo

4 Monad

todo

5 Arrow operator as Functor, Applicative, Monad

let `F = (->) r = r ->`

5.1 Functor

`((->) r) = (r ->)` as a functor
`(r ->) *` expects a type as argument
instances of `(->) r *` as a type class
examples of other functors: `[] *`, `Maybe *`
functor as a type constructor

fmap:

```
<$>:: (a -> b) -> F a -> F b
<$>:: (a -> b) -> ((->) r) a -> ((->) r) b
<$>:: (a -> b) -> (r -> a) -> (r -> b)
<$>:: (a -> b) -> (r -> a) -> r -> b
```

composition operator:

```
 (.) :: (b -> c) -> (a -> b) -> a -> c
```

therefore,
`<$> = (.)` where `F = (->) r`

5.1.1 example

```
(+) <$> (*2)
(+) . (*2)
\x -> (+) ((*2) x)
\x -> (+) (x*2)
\x -> x*2 :: a -> a
(\x -> (+) (x*2)) :: a -> (a -> a)
(\x -> (+) (x*2)) :: a -> a -> a
(\x -> ((x*2)+) :: a -> a -> a
(\x -> (\y -> (x*2) + y) :: a -> a -> a
```

5.2 Applicative

apply:

```
<*>:: F (a -> b) -> F a -> F b
<*>:: ((->) r) (a -> b) -> ((->) r) a -> ((->) r) b
<*>:: (r -> a -> b) -> (r -> a) -> (r -> b)
h <*> h = \r -> g r (h r)
```

where `g :: r -> a -> b`
where `h :: r -> a`

```
pure :: a -> F a
pure :: a -> (->) r a
pure :: a -> r -> a
pure = const
```

5.2.1 example

```
(+) <$> (*2) <*> (+10)
(+) . (*2) <*> (+10)
(\x -> (+) (x*2)) <*> (\x -> x + 10)
\x -> (+) (x*2) (x+10)
```

types:
`\x -> :: (->) r`
`(+) (x*2) :: a -> b` where `x` is fixed
`\x -> (+) (x*2) :: ((->) r) a -> b`

`\x -> x + 10 :: r -> a  = ((->) r) a`

`(+) (x*2) (x+10) :: b` where `x` is fixed
`\x -> (+) (x*2) (x+10) :: ((->) r) b`

```
<*> :: (((->) r) a -> b) -> (((->) r) a) -> (((->) r) b)
```

thus types are as expected for applicative

5.3 Monad

```
return :: a -> F a
return :: a -> (->) r a = a -> r -> a
const :: a -> r -> a
return = const
```

bind:

```
>>= :: F a -> (a -> F b) -> F b
>>= :: (->) r a -> (a -> (->) r b) -> (->) r b
>>= :: (r -> a) -> (a -> r -> b) -> r -> b
 g >>= h = \r -> h (g r) r
```

where `g :: (r -> a)`
where `h :: (a -> r -> b)`

```
 g >>= h = \r -> (\x y z -> x z y) h (g r) r
 g >>= h = \r -> (flip  h) r (g r)
 g >>= h = (flip  h) <*> g
```

References

[1] Allen & Moronuki. Haskell programming from first principles, 2016.