



Meetup



Friendly Environment Policy



Berlin Code of Conduct



**CATEGORY THEORY
FOR PROGRAMMERS**



Bartosz Milewski

**Category
Theory
for
Programmers
Chapter 20-22:
Monads**

| | | |
|-----------|--|------------|
| 20 | Monads: Programmer's Definition | 289 |
| 20.1 | The Kleisli Category | 291 |
| 20.2 | Fish Anatomy | 294 |
| 20.3 | The do Notation | 296 |
| | | |
| 21 | Monads and Effects | 301 |
| 21.1 | The Problem | 301 |
| 21.2 | The Solution | 302 |
| 21.2.1 | Partiality | 303 |
| 21.2.2 | Nondeterminism | 304 |
| 21.2.3 | Read-Only State | 307 |
| 21.2.4 | Write-Only State | 308 |

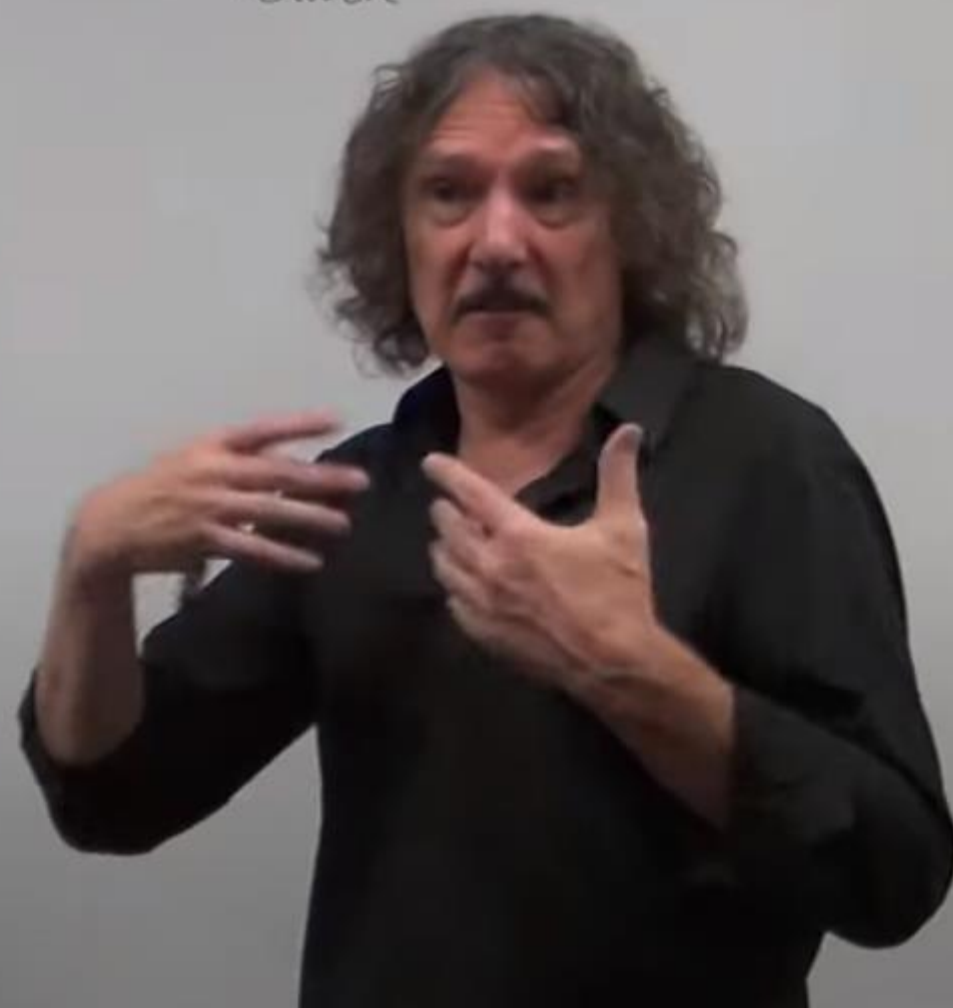
| | | |
|--------|------------------------------|-----|
| 21.2.5 | State | 309 |
| 21.2.6 | Exceptions | 311 |
| 21.2.7 | Continuations | 311 |
| 21.2.8 | Interactive Input | 313 |
| 21.2.9 | Interactive Output | 316 |
| 21.3 | Conclusion | 317 |

22 Monads Categorically 318

| | | |
|------|---|-----|
| 22.1 | Monoidal Categories | 323 |
| 22.2 | Monoid in a Monoidal Category | 329 |
| 22.3 | Monads as Monoids | 331 |
| 22.4 | Monads from Adjunctions | 333 |

•
id

>=>
return



```
class Monad m where
```

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

```
    return :: a -> m a
```

```
class Monad m where
    (==>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
    return :: a -> m a
```

For every monad, instead of defining the fish operator, we may instead define bind. In fact the standard Haskell definition of a monad uses bind:

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```



```
class Monad m where
    (==>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
    return :: a -> m a
```

For every monad, instead of defining the fish operator, we may instead define bind. In fact the standard Haskell definition of a monad uses bind:

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

That leads us to the third option for defining a monad:

```
class Functor m => Monad m where
    join :: m (m a) -> m a
    return :: a -> m a
```

```
instance Monad [] where  
  join = concat  
  return x = [x]
```

In the list monad — Haskell's implementation of nondeterministic computations — `join` is implemented as `concat`. Remember that `join` is supposed to flatten a container of containers — `concat` concatenates a list of lists into a single list. `return` creates a singleton list:

```
instance Monad [] where
    join = concat
    return x = [x]
```

```
instance Monad Maybe where
  Nothing >>= k = Nothing
  Just a   >>= k = k a
  return a = Just a
```

```
import Control.Monad
import Data.Maybe

-- not generic to avoid error
safeHead :: [Int] -> Maybe Int
safeHead [] = Nothing
safeHead (x:_) = Just x

safeDouble :: Int -> Maybe Int
safeDouble x = Just (2 * x)

double :: Int -> Int
double = (2*)

main :: IO ()
main = do
    print $ safeHead [42,1729,343]
    print $ safeHead []
    print $ (safeHead ==> safeDouble) [42,1729,343]
    print $ fmap double $ safeHead [42,1729,343]
    print $ (fmap double . safeHead) [42,1729,343]
    print $ double <$> (safeHead [42,1729,343])
    print $ (<$>) double (safeHead [42,1729,343])
    print $ (safeHead ==> (return . double)) [42,1729,343]
    print $ (Just [42,1729,343]) >=> safeHead >=> safeDouble
    print $ (safeHead [42,1729,343]) >=> safeDouble
```



Meetup