ADVANCED

# FUNCTIONAL PROGRAMMING

## OUTLINE

- Organization
- Intro to FP: Recap
- Monad stacks

## MEET YOUR INSTRUCTOR

- Ekaterina Verbitskaia
- Researcher @ JetBrains Research PLAN, Programming Languages and Program Analysis Lab
- Interested in:
  - Functional and Relational Programming
  - Partial Evaluation, Metacomputations
- @kajigor

## ORGANIZATION

- 2 classes per week
- Written exam (50% of your mark)
- Work during the term (50%  of your mark)
  - Homework assignments (40/100 points)
  - 75-minutes presentation on a research topic of your choice (40/100 points)
  - Active participation during classes (40/100 points)

## WHAT YOU NEED TO START

- Install with GHCup:
  - GHC 9.6.6 – compiler
  - Stack 3.1.1 – build system
  - HLS 2.9.0.1 – language server
- VSCode
  - Haskell extension

## OUTLINE

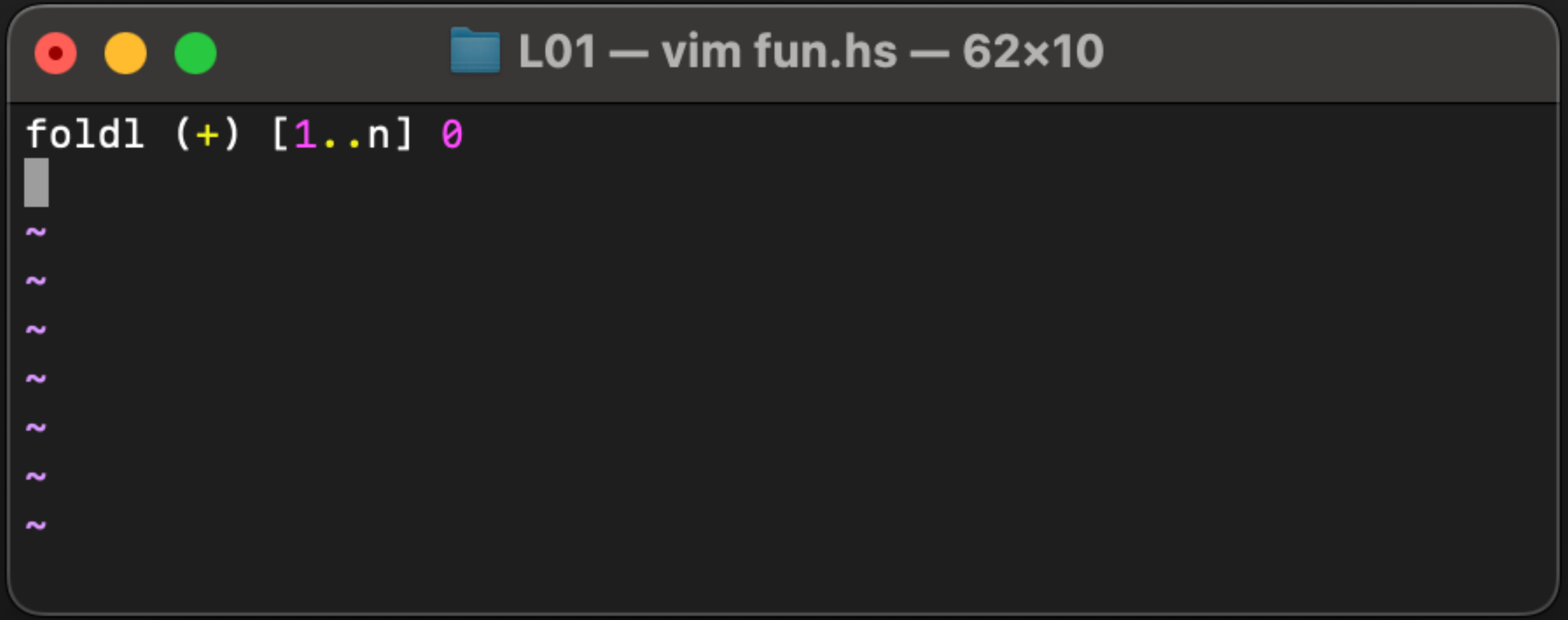
- Organization
- Intro to FP: Recap
- Monad stacks

## WHAT IS FUNCTIONAL PROGRAMMING?

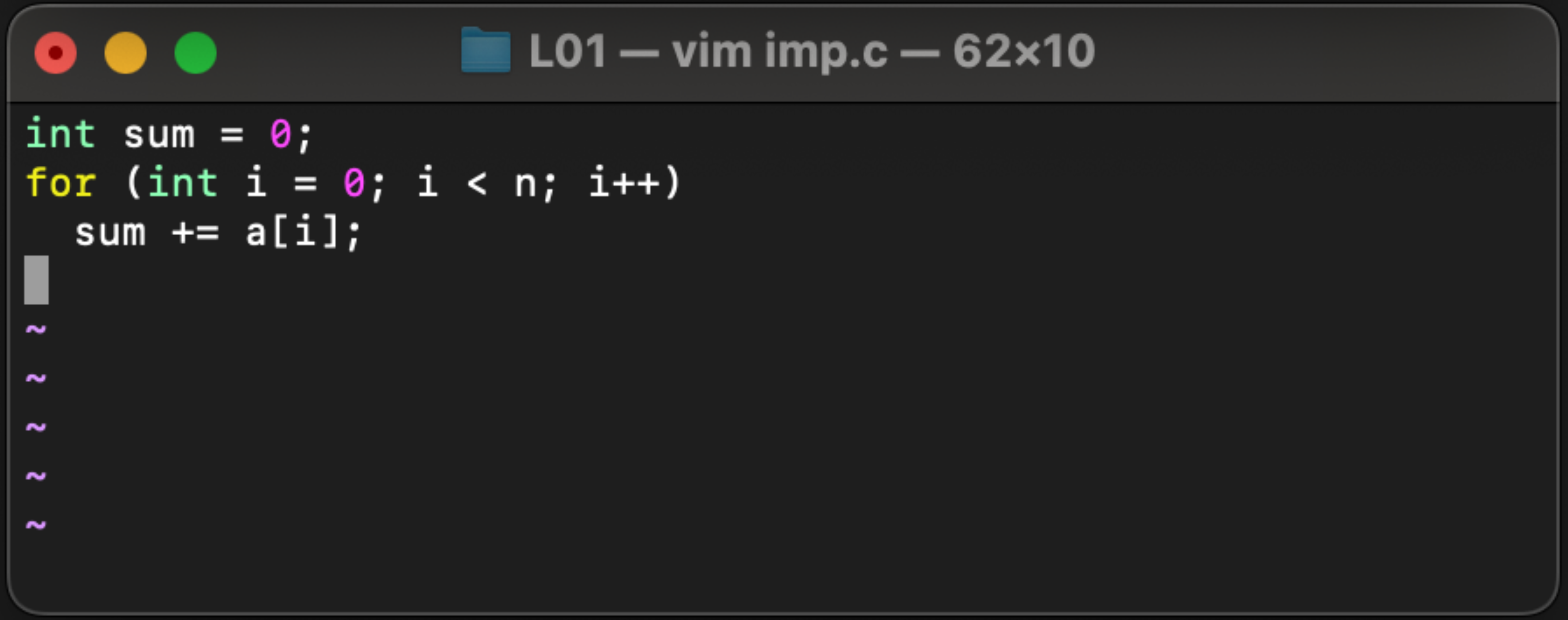
## WHAT IS FUNCTIONAL PROGRAMMING?

- Functional programming computes by means of evaluation of functions

- Imperative programming uses statements to change the program's state



```
foldl (+) [1..n] 0
```

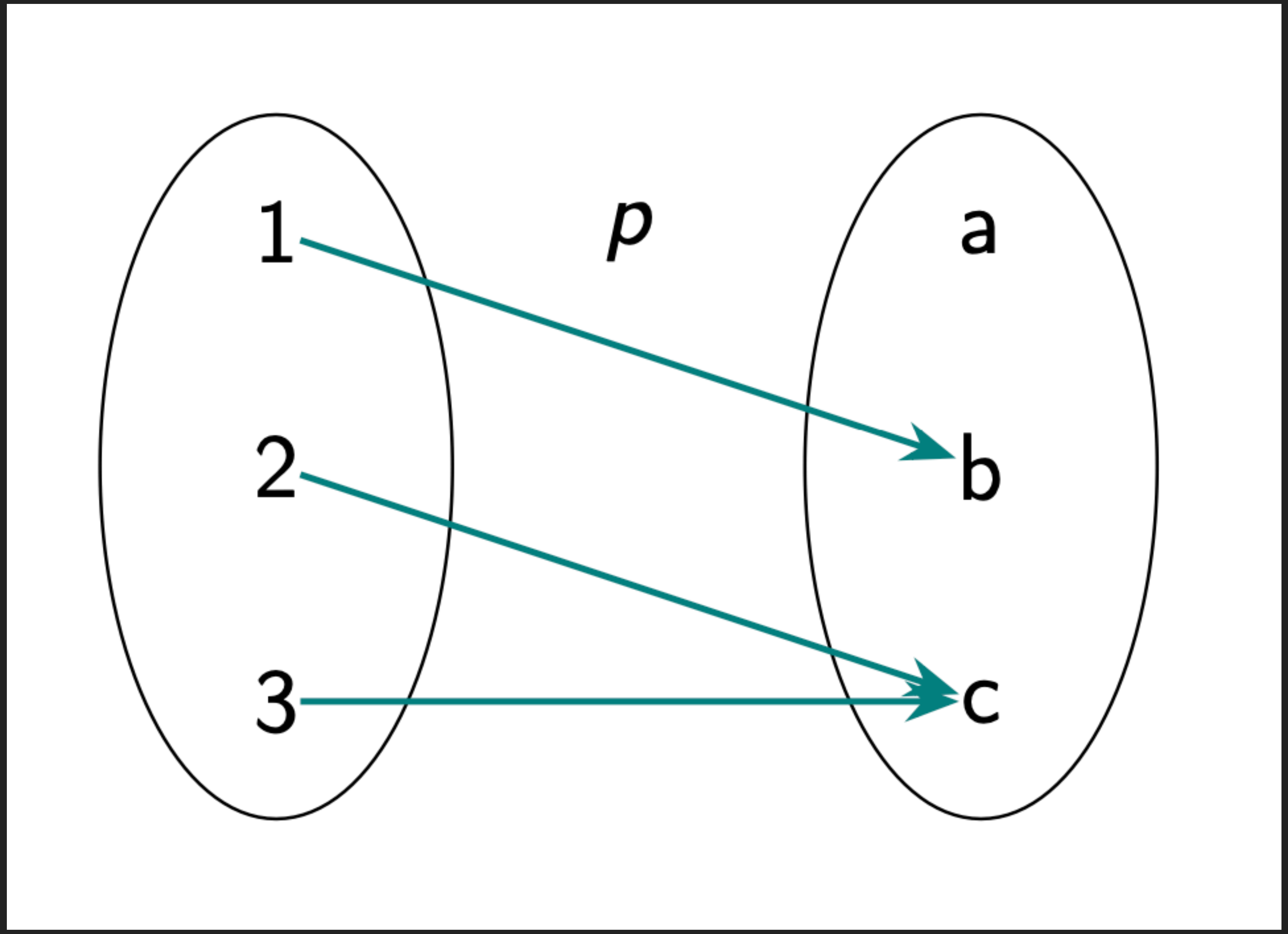


```
int sum = 0;
for (int i = 0; i < n; i++)
  sum += a[i];
```

## WHAT IS A FUNCTION?

## WHAT IS A FUNCTION?

- Fundamental building block of a program
- Pure
  - No side-effects such as IO or mutable state
  - **Always** the same result for the same input
- First-class: functions can be passed as arguments and returned as results
- Composition to create new functions

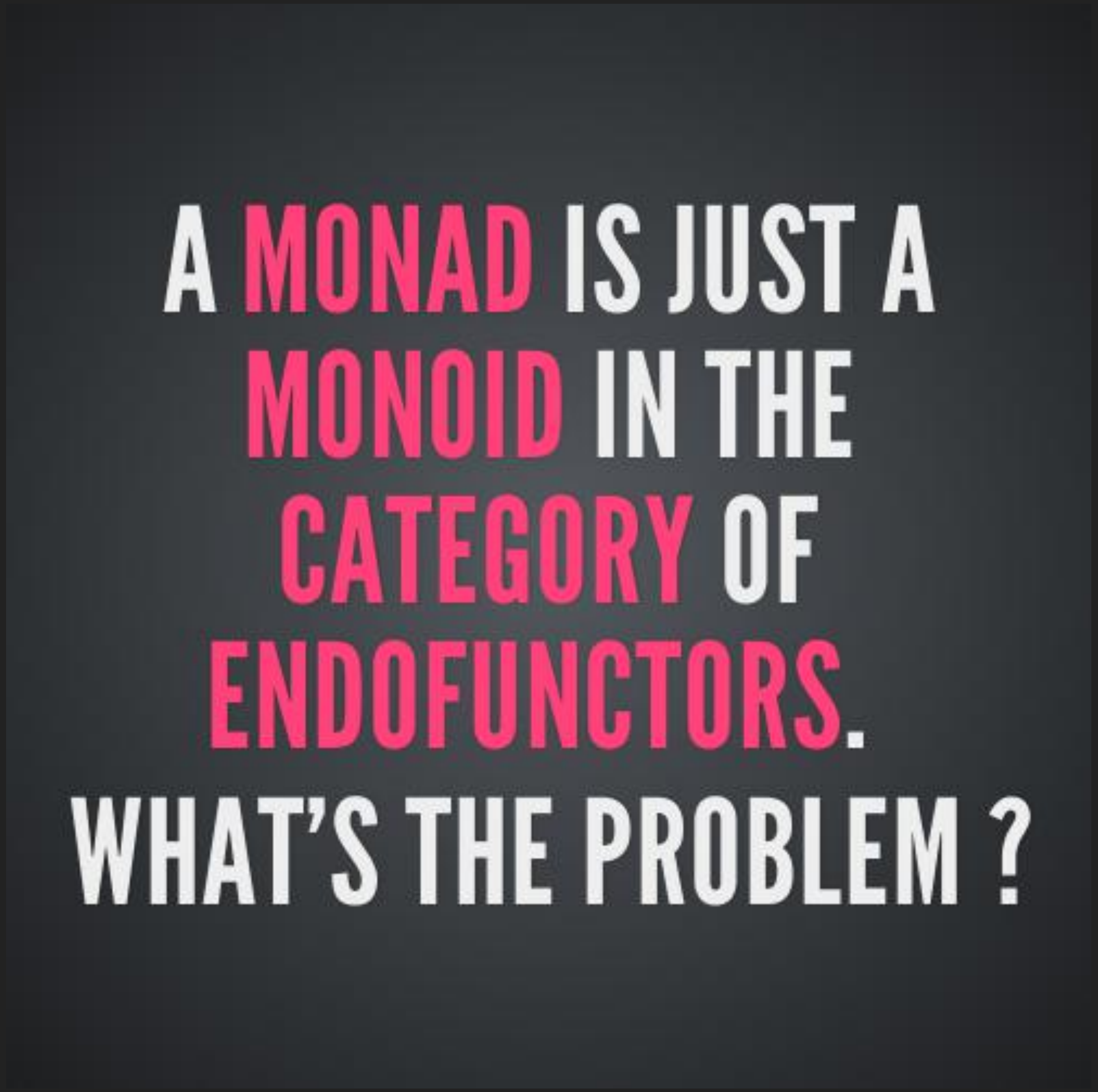


## IS MAIN A PURE FUNCTION?

## IS MAIN A PURE FUNCTION?

- No
- `main :: IO ()`
  - Does not return any (useful) value
  - We are only interested in its effect

## WHAT IS A MONAD?

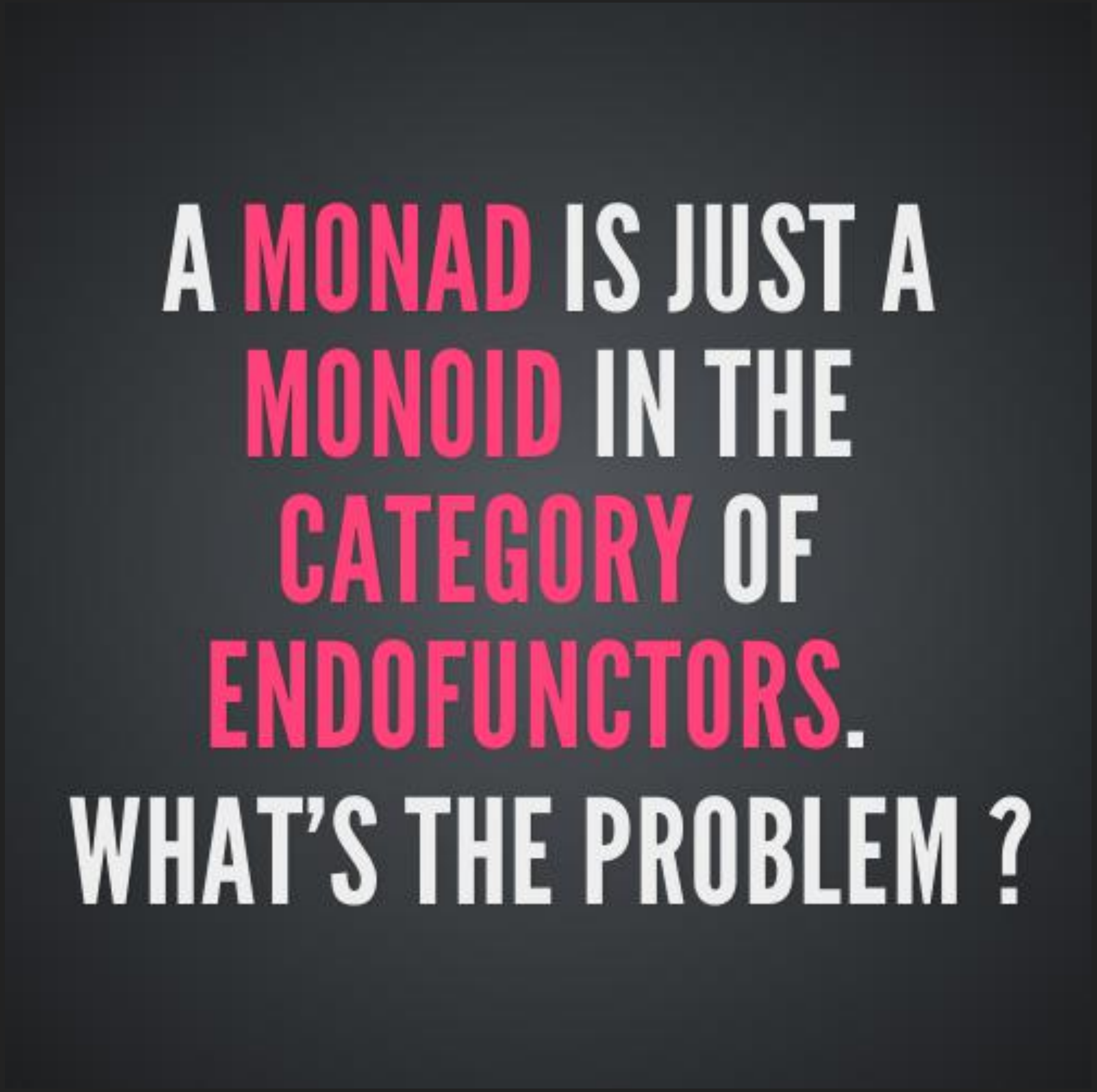


A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTORS. WHAT'S THE PROBLEM ?

## WHAT IS A MONAD?

```haskell
class Applicative m => Monad (m :: Type -> Type) where
  (>>=) :: forall a b. m a -> (a -> m b) -> m b

  (>>) :: forall a b. m a -> m b -> m b
  m >> k = m >>= \_ -> k

  return :: a -> m a
  return = pure
```

```haskell
return a >> k = k a
m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h

pure = return
m1 <*> m2 = m1 >>= \x1 -> m2 >>= \x2 -> return (x1 x2)

fmap f xs = xs >>= return . f
```



A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTORS.
WHAT'S THE PROBLEM ?

## WHY DO WE USE MONADS?

- To represent impure computations
  - Errors
  - Non-determinism
  - State
  - IO
- To represent sequential computation
- **To hide boilerplate behind do-notation**

## ERRORS

## ERRORS

- `Maybe a`
  - Use when there is a single way to fail
- `Either e a`
  - Use a custom error type to describe what went wrong

```haskell
instance Monad Maybe where
  Just x >>= f = f x
  Nothing >>= _ = Nothing

safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just $ x `div` y

chainDiv :: [Int] -> Maybe Int
chainDiv [] = Nothing
chainDiv (h : t) = foldM safeDiv h t
```

## NONDETERMINISM

## NONDETERMINISM

- `[a]`
  - The default implementation features depth-first search

```haskell
instance Monad [] where
  xs >>= f = concat (map f xs)
```

## NONDETERMINISM

- [a]
  - The default implementation features depth-first search

```
instance Monad [] where
  xs >>= f = concat (map f xs)

pairs =
  [ (x, y)
  | x <- ['a' .. 'z'] <- won't progress
  | y <- [0 ..]       <- until exhausted
  ]
```

## NONDETERMINISM

- `[a]`
  - The default implementation features depth-first search
- `Logic a`
  - Paper
  - Interleaving search

```
instance Monad [] where
  xs >>= f = concat (map f xs)

pairs =
  [ (x, y)
  | x <- ['a' .. 'z'] <- won't progress
  | y <- [0 ..]       <- until exhausted
  ]
```

## ENVIRONMENTS, LOGGING, STATE

## ENVIRONMENTS, LOGGING, STATE

- `State s a`
  - Functions as `Reader r a` + `Writer w a`
  - Convenient interface to manipulate mutable state

```haskell
newtype State s a
  = State { runState :: s → (s, a) }

instance Monad (State s) where
  m >>= k = State $ \s →
    let (s', x) = runState m s in
    runState (k x) s'

get :: State s s
get = State $ \s → (s, s)

put :: s → State s ()
put s = State $ \_ → (s, ())

modify :: (s → s) → State s ()
modify f = do
  s ← get
  put (f s)
```

## IO MONAD

- Implementation is hidden
- You can think of it as a State monad over some RealWorld representation

```
newtype IO a
  = IO { runIO :: RealWorld
                → (RealWorld, a)}
```

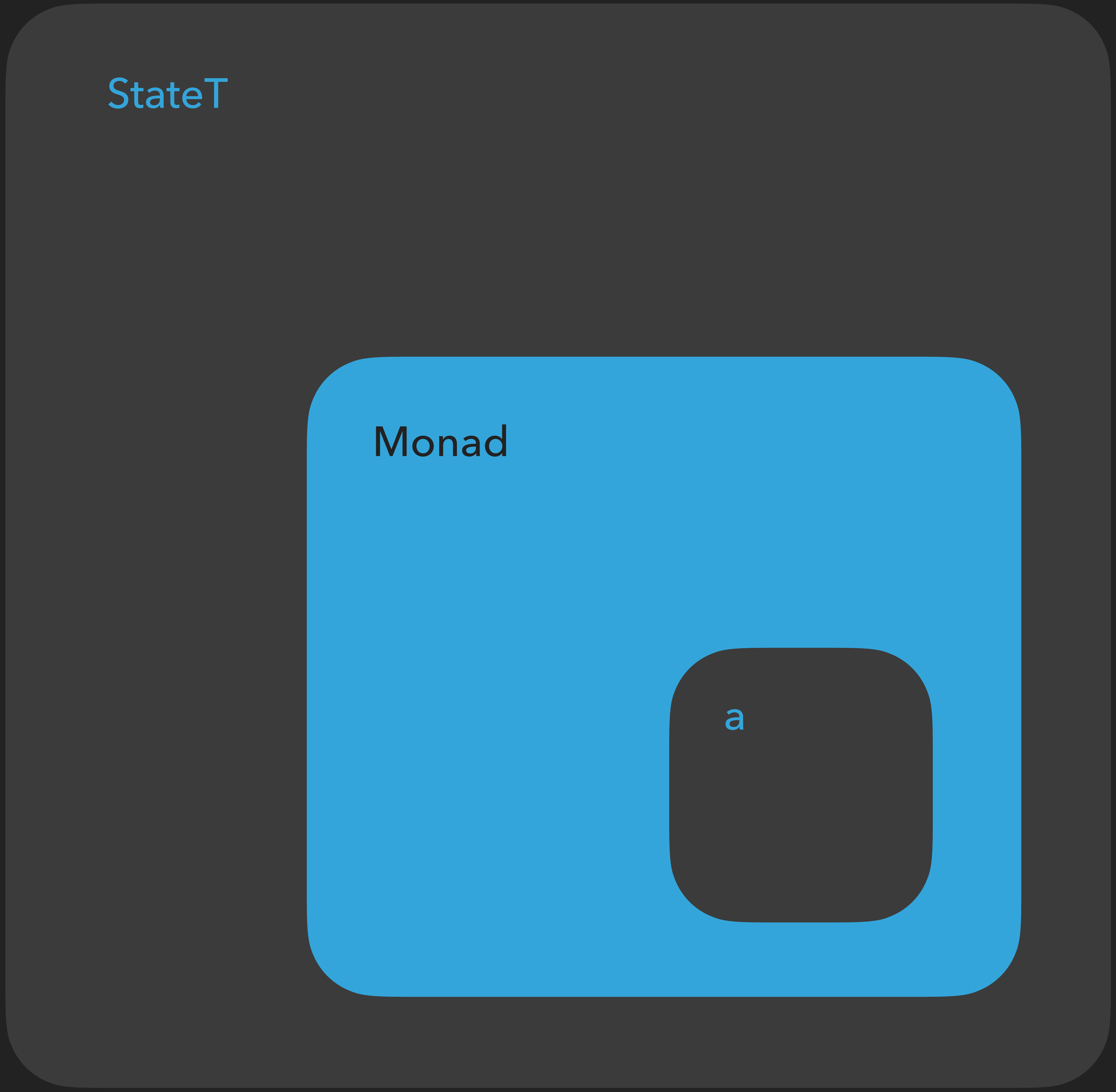
## EXERCISE

- Implement an interpreter for the formulas in Reverse Polish Notation (see RPN.hs)

```
ghci> eval "1 2 +"
3
ghci> eval "1 2 *"
2
ghci> eval "1 2 -"
-1
ghci> eval "1 2 + 3 * 4 -"
5
```

## HOW CAN WE COMBINE EFFECTS?

- Our `eval` function can sometimes fail, but it also needs to have access to mutable state
- Let's combine the two monads into a **stack**
- Which `>>=` do we use?



## OUTLINE

- Organization
- Intro to FP: Recap
- Monad stacks

## MONAD TRANSFORMER

- Should define a monad
- Should provide a way to use the features of both monads
- Should implement `lift`
    - Don't forget about the laws

```
class (forall m. Monad m ⇒ Monad (t m))
  ⇒ MonadTrans t where
  lift :: Monad m ⇒ m a → t m a

  -- lift . return = return
  -- lift (m >>= f) = lift m >>= (lift . f)
```

## THE ORDER OF MONADS IN A STACK

- `StateT Maybe a`
  - `~ s → Maybe (a, s)`
  - We lose both state and the result in case of an error
- `MaybeT (State s)`
  - `~ s → (Maybe a, s)`
  - Error only in the result; state survives

```haskell
class (forall m. Monad m ⇒ Monad (t m))
  ⇒ MonadTrans t where
  lift :: Monad m ⇒ m a → t m a

  -- lift . return = return
  -- lift (m >>= f) = lift m >>= (lift . f)
```

## EXERCISE

- Implement `MyMaybeT` – monad transformer for `Maybe`
- Redefine the evaluator for RPN to use `MyMaybeT` as the outer monad

```haskell
class (forall m. Monad m ⇒ Monad (t m))
  ⇒ MonadTrans t where
  lift :: Monad m ⇒ m a → t m a

  -- lift . return = return
  -- lift (m ≫= f) = lift m ≫= (lift . f)

newtype MyMaybeT m a =
  MyMaybeT { runMyMaybeT :: m (Maybe a) }
```

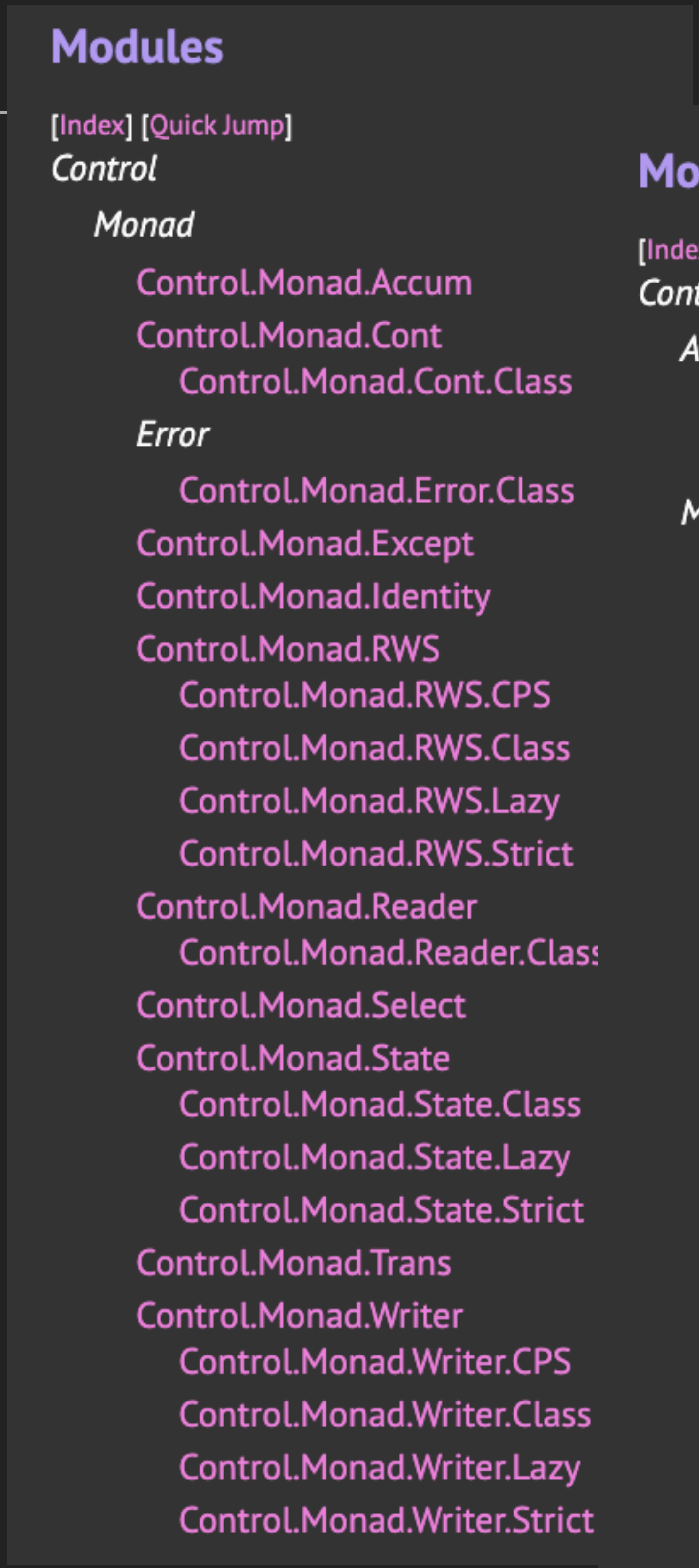
## LIFTING IS EXHAUSTING

- Using `lift` means we delegate some functionality to another monad
    - `lift (lift m)`
    - `lift (lift (lift m))`
- This is fragile if we change the monad stack
- Solution: use monad-specific interfaces
    - Any `MonadState` method within a monad stack build with `MaybeT` is silently lifted

```haskell
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  get = state (\s → (s, s))

  put :: s → m ()
  put s = state (\_ → ((), s))

  state :: (s → (a, s)) → m a

instance MonadState s m ⇒
  MonadState s (MaybeT m)  where

  state = lift . state
```

## COMMON TRANSFORMERS

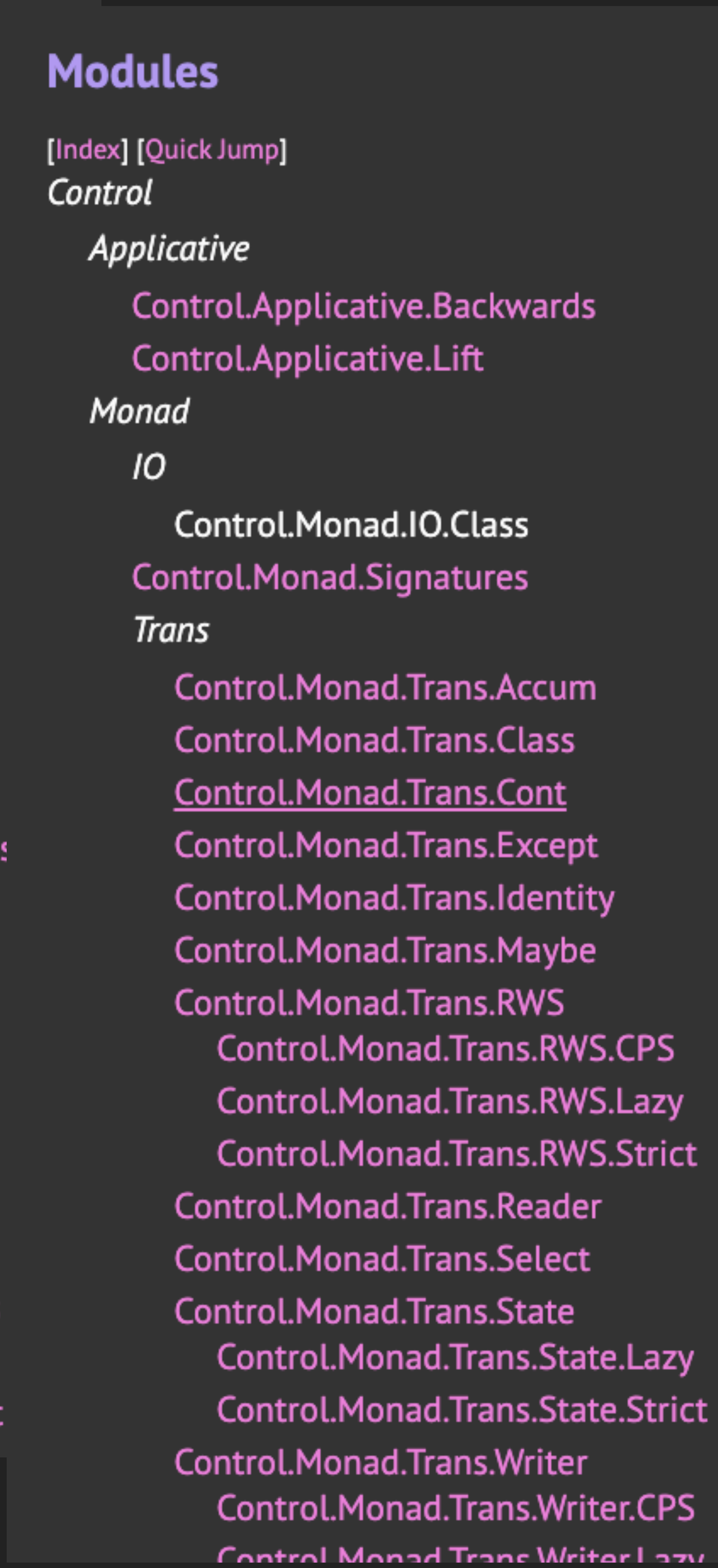
- Packages
  - mtl
  - transformers
- `IdentityT`
- `MaybeT, ExceptT`
- `ReaderT, WriterT, StateT, RWST`
- `AccumT`
- `ContT`



**Modules**

[Index] [Quick Jump]

- *Control*
  - *Monad*
    - Control.Monad.Accum
    - Control.Monad.Cont
      - Control.Monad.Cont.Class
    - *Error*
      - Control.Monad.Error.Class
    - Control.Monad.Except
    - Control.Monad.Identity
    - Control.Monad.RWS
      - Control.Monad.RWS.CPS
      - Control.Monad.RWS.Class
      - Control.Monad.RWS.Lazy
      - Control.Monad.RWS.Strict
    - Control.Monad.Reader
      - Control.Monad.Reader.Clas
    - Control.Monad.Select
    - Control.Monad.State
      - Control.Monad.State.Class
      - Control.Monad.State.Lazy
      - Control.Monad.State.Strict
    - Control.Monad.Trans
    - Control.Monad.Writer
      - Control.Monad.Writer.CPS
      - Control.Monad.Writer.Class
      - Control.Monad.Writer.Lazy
      - Control.Monad.Writer.Strict



**Modules**

[Index] [Quick Jump]

- *Control*
  - *Applicative*
    - Control.Applicative.Backwards
    - Control.Applicative.Lift
  - *Monad*
    - *IO*
      - Control.Monad.IO.Class
    - Control.Monad.Signatures
    - *Trans*
      - Control.Monad.Trans.Accum
      - Control.Monad.Trans.Class
      - Control.Monad.Trans.Cont
      - Control.Monad.Trans.Except
      - Control.Monad.Trans.Identity
      - Control.Monad.Trans.Maybe
      - Control.Monad.Trans.RWS
        - Control.Monad.Trans.RWS.CPS
        - Control.Monad.Trans.RWS.Lazy
        - Control.Monad.Trans.RWS.Strict
      - Control.Monad.Trans.Reader
      - Control.Monad.Trans.Select
      - Control.Monad.Trans.State
        - Control.Monad.Trans.State.Lazy
        - Control.Monad.Trans.State.Strict
      - Control.Monad.Trans.Writer
        - Control.Monad.Trans.Writer.CPS
        - Control.Monad.Trans.Writer.Lazy

## EXCEPTT

- Did you miss exceptions?
- Now you can have them in Haskell

```haskell
newtype ExceptT e m a =
  ExceptT (m (Either e a))

throwE :: Monad m ⇒ e → ExceptT e m a

catchE ::
    Monad m
    ⇒ ExceptT e m a
    → (e → ExceptT e' m a)
    → ExceptT e' m a

handleE ::
    Monad m
    ⇒ (e → ExceptT e' m a)
    → ExceptT e m a
    → ExceptT e' m a
```

## ACCUMT

- Limited version to `StateT`
  - Only uses `<>` to modify state
- Or `Writer` with the additional ability to see the result of previous `tells`
- Works faster than the default `StateT`

```haskell
newtype AccumT w m a = AccumT (w → m (a, w))

look :: m w

add :: w → m ()

accum :: (w → (a, w)) → m a
```