(Alternatives to) Lazy I/O

Edsko de Vries

Well-Typed
The Haskell Consultants

Warmup

```
shout :: FilePath -> FilePath -> IO ()
shout inPath outPath = do
    inh  <- openFile inPath  ReadMode
    outh <- openFile outPath WriteMode
    go inh outh
    hClose inh
    hClose outh
  where
    go inh outh = do
      eof <- hIsEOF inh
      unless eof $ do
        line <- hGetLine inh
        hPutStrLn outh (map toUpper line)
        go inh outh
```

Resource reclamation

```
shout :: FilePath -> FilePath -> IO ()
shout inPath outPath =
    withFile inPath  ReadMode  $ \inh  ->
    withFile outPath WriteMode $ \outh ->
      go inh outh
  where
    go inh outh = do
      eof <- hIsEOF inh
      unless eof $ do
        line <- hGetLine inh
        hPutStrLn outh (map toUpper line)
        go inh outh
```

Shout using Lazy I/O

```
shout :: FilePath -> FilePath -> IO ()
shout inPath outPath = do
  contents <- readFile inPath
  let contents' = map toUpper contents
  writeFile outPath contents'
```

unsafeInterleaveIO

$$\text{forall } (x\ y :: \text{Int}).\ x + y = y + x$$

Haddock

unsafeInterleaveIO :: IO a -> IO a

unsafeInterleaveIO allows IO computation to be deferred lazily. When passed a value of type IO a, the IO will only be performed *when* the value of the a is *demanded*.

```
ex4ref :: IORef Int
{-# NOINLINE ex4ref #-}
ex4ref = unsafePerformIO $ newIORef 0

ex4_mainA :: IO ()
ex4_mainA = do
  x <- unsafeInterleaveIO $ readIORef ex4ref
  y <- unsafeInterleaveIO $ writeIORef ex4ref 1 >> return 5
  print (x + y)
```

Referentially opaque context

Better mental model

unsafeInterleaveIO :: IO a -> IO a

unsafeInterleaveIO allows IO computation to be deferred lazily. When passed a value of type IO a, the IO will be performed *at some point* between the call to unsafeInterleaveIO and the demand of the value.

```
ex4ref :: IORef Int
{-# NOINLINE ex4ref #-}
ex4ref = unsafePerformIO $ newIORef 0

ex4_mainA :: IO ()
ex4_mainA = do
  x <- unsafeInterleaveIO $ readIORef ex4ref
  y <- unsafeInterleaveIO $ writeIORef ex4ref 1 >> return 5
  print (x + y)
```

Lazy I/O (naive version)

```
hGetContents :: Handle -> IO String
hGetContents h = unsafeInterleaveIO $ do
   eof <- hIsEOF h
   if eof then return []
          else do c  <- hGetChar h
                  cs <- hGetContents h
                  return (c:cs)
```

```
ex5_mainA :: IO ()
ex5_mainA = do
  h <- openFile "hello.txt" ReadMode
  hGetContents h >>= mapM_ putChar
  hClose h
```

```
ex5_mainB :: IO ()
ex5_mainB = do
  h <- openFile "hello.txt" ReadMode
  contents <- hGetContents h
  hClose h
  print (length contents)
```

```
ex5_mainC :: IO ()
ex5_mainC = do
  h <- openFile "hello.txt" ReadMode
  contents <- hGetContents h
  let len = length contents
  hClose h
  print len
```

```
ex5_mainD :: IO ()
ex5_mainD = do
  h <- openFile "hello.txt" ReadMode
  contents <- hGetContents h
  len <- evaluate $ length contents
  hClose h
  print len
```

```
hGetContents' :: Handle -> IO String
hGetContents' h = unsafeInterleaveIO $ do
  eof <- hIsEOF h
  if eof then hClose h >> return []
         else do c  <- hGetChar h
                 cs <- hGetContents' h
                 return (c:cs)
```

```
ex6_mainB :: IO ()
ex6_mainB = do
  h <- openFile "hello.txt" ReadMode
  contents <- hGetContents' h
  mapM_ putChar (take 10 contents)
```

openFile adds a finalizer to the handle which will close the associated file descriptor

```
ex6_main :: IO ()
ex6_main = do
  h <- openFile "hello.txt" ReadMode
  contents <- hGetContents' h
  mapM_ putChar (take 10 contents)
```

contents <- readFile “hello.txt”

```
tar <- Server.exportServerTar server
-- It is EXTREMELY IMPORTANT that we force the tarball to be
-- constructed now. If we wait until it is demanded in the next
-- withServer context then the tar gets filled with entirely
-- wrong files!
BS.length tar `seq` return (tar, test_roundtrip)
```

(Real code fragment)

```
forkM_maybe :: SDoc -> IfL a -> IfL (Maybe a)
-- Run thing_inside in an interleaved thread.
-- It shares everything with the parent thread, so this is DANGEROUS.
--
-- It returns Nothing if the computation fails
--
-- It's used for lazily type-checking interface
-- signatures, which is pretty benign
forkM_maybe doc thing_inside
 = do { unsafeInterleaveM $ ...
```

(Real code fragment)

Lazy I/O summary

- Can use standard pure code (such as normal operations on Strings)
- Have to be very careful to force values to be evaluated
- Timely resource reclamation is difficult

io-streams

```
data InputStream a = InputStream {
      _read   :: IO (Maybe a)
    , _unRead :: a -> IO ()
    }

data OutputStream a = OutputStream {
      _write :: Maybe a -> IO ()
    }

makeInputStream :: IO (Maybe a) -> IO (InputStream a)
```

```
connect :: InputStream a -> OutputStream a -> IO ()
connect p q = loop
  where
    loop = do
      m <- read p
      write m q
      case m of
        Nothing -> return ()
        Just _  -> loop
```

```
shout :: FilePath -> FilePath -> IO ()
shout inPath outPath =
  withFileAsInput  inPath  $ \ins  ->
  withFileAsOutput outPath $ \outs -> do
    ins' <- S.map (BS.map toUpper) ins
    connect ins' outs
```

```haskell
map :: (a -> b) -> InputStream a -> IO (InputStream b)
map f s = makeInputStream g
  where
    g = read s >>= return . fmap f
```

:: Maybe a

```haskell
data DupState a = Waiting | Dup a | Done

dup :: InputStream a -> IO (InputStream a)
dup s = do
  stRef <- newIORef Waiting
  makeInputStream $ do
    st <- readIORef stRef
    case st of
      Waiting -> do ma <- S.read s
                    case ma of
                      Nothing -> writeIORef stRef Done
                      Just a  -> writeIORef stRef (Dup a)
                    return ma
      Dup a   -> do writeIORef stRef Waiting
                    return (Just a)
      Done    -> return Nothing
```

cf. using pipes:

```haskell
dup = forever $ do
  x <- request
  respond x
  respond x
```

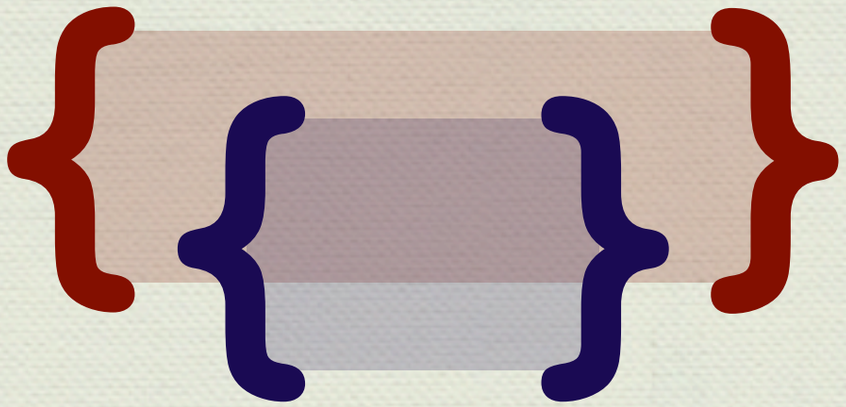
```
ghci> old <- S.fromList [1, 2, 3]
ghci> new <- S.map id old
ghci> S.read new
Just 1
ghci> S.read old
Just 2
ghci> S.read new
Just 3
```

```
ghci> old <- S.fromList [1, 2, 3]
ghci> new <- S.map id old
ghci> S.read new
Just 1
ghci> S.unRead (fromJust it) new
ghci> S.read old
Just 2
```

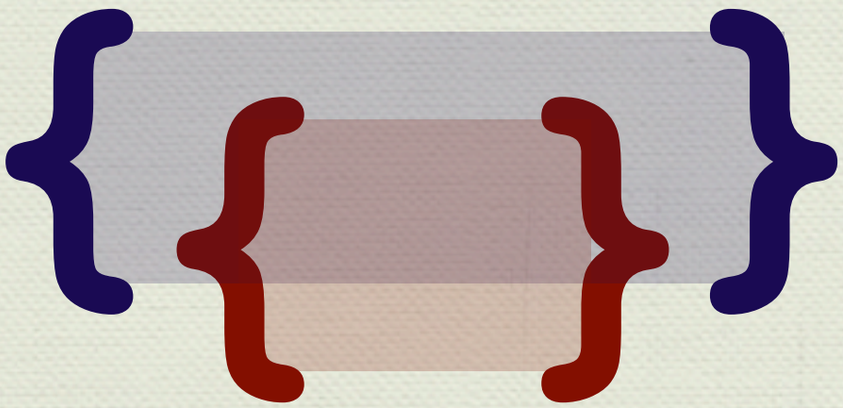
S.read new would give (Just 1)

Limitations of `withXXX`

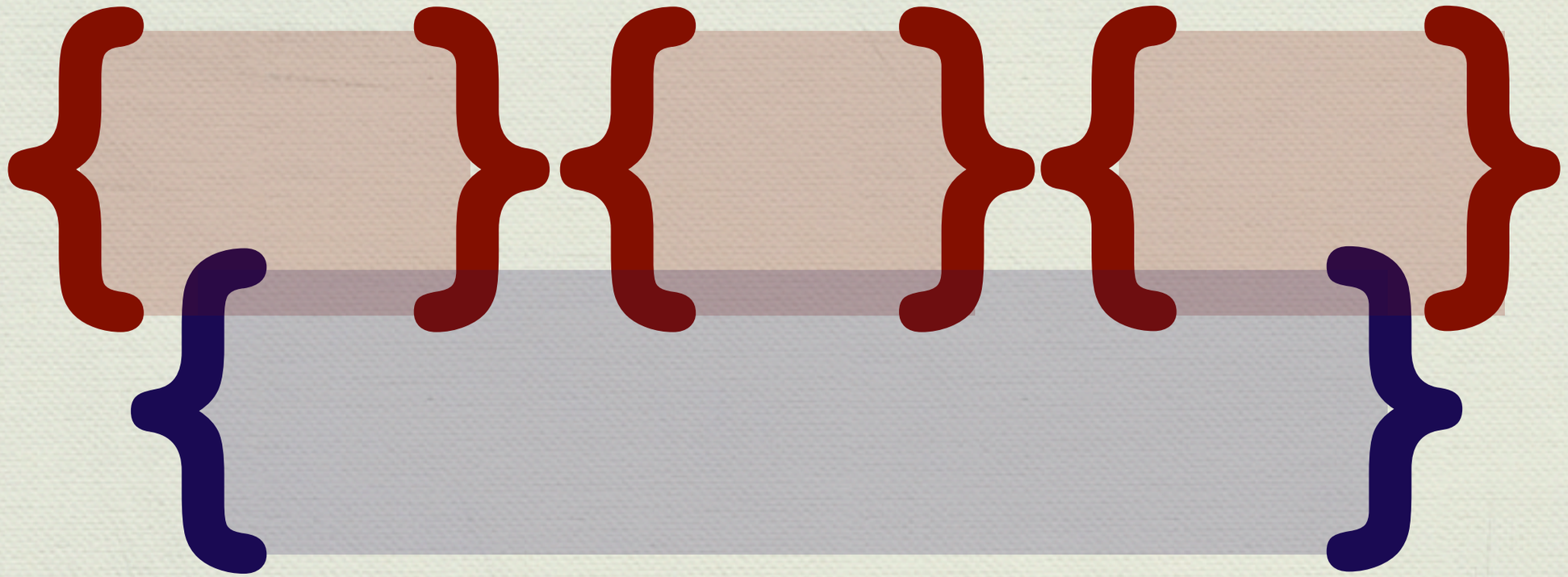
```
withFile inPath ReadMode $ \inh ->
  withFile outPath WriteMode $ \outh ->
    -- do something with inh and outh
```



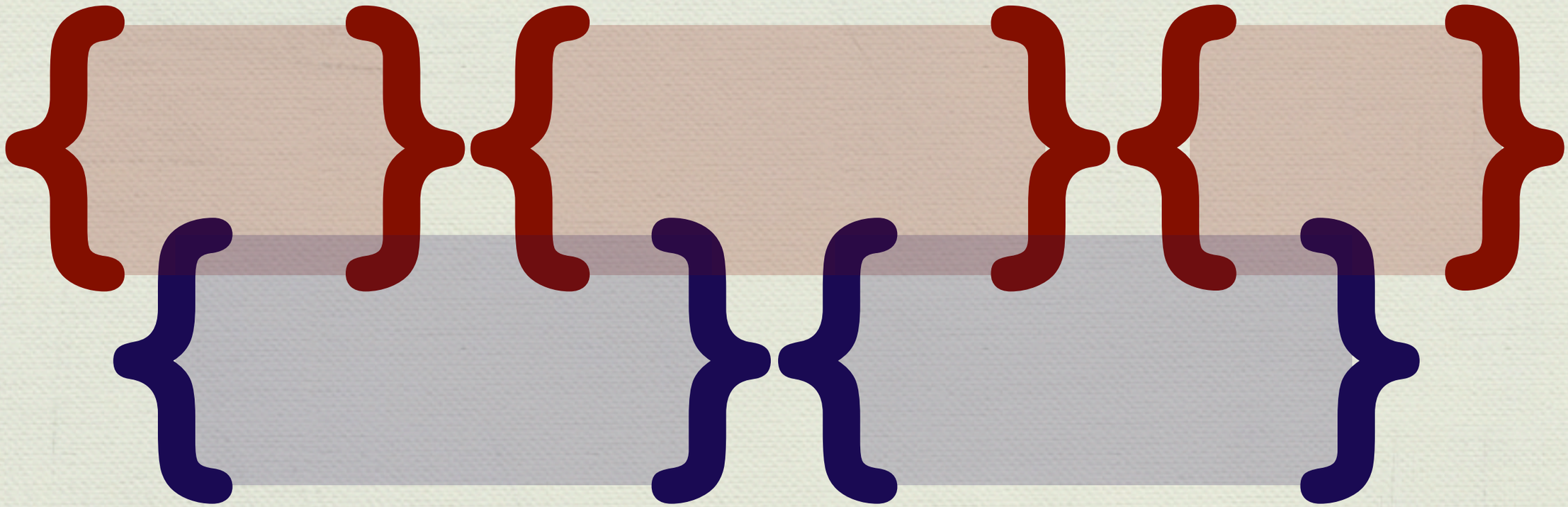
```
withFile outPath WriteMode $ \outh ->
  withFile inPath ReadMode $ \inh ->
    -- do something with inh and outh
```



Limitations of `withXXX`



Limitations of `withXXX`



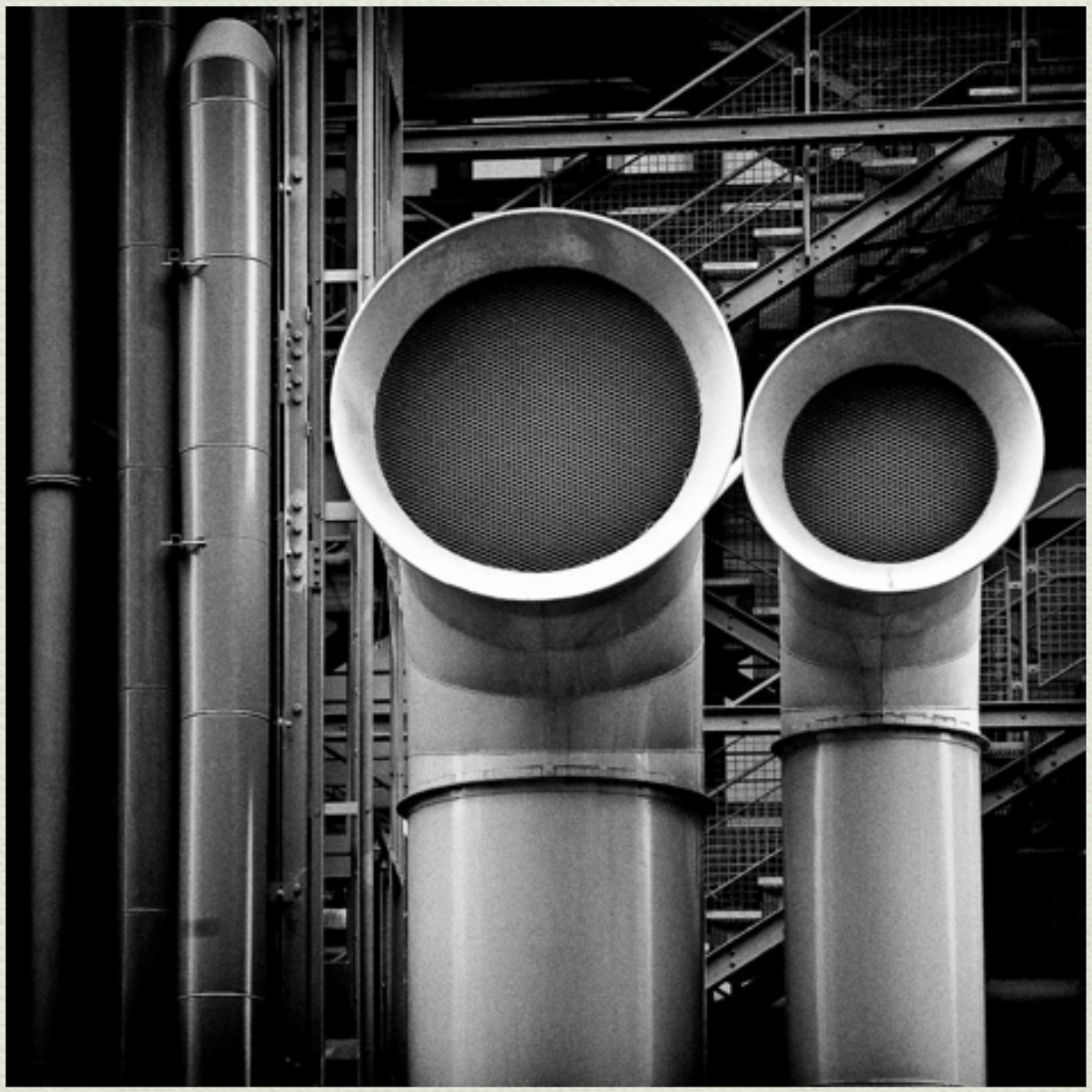
io-streams: criticisms

- Block-scoped (`withXXX` style)
- Programming with explicit state (*cf.* "`dup`")
- unRead is non-compositional (`map id p /= return p`)

pipes / conduits

Pipes From Scratch

- **Requests**
- Responses
- Basic pipes
- Finalizers
- Exceptions
- Asynchronous exceptions
- Leftovers



Pipes: Requests (1)

```
examplePipe :: Pipe Int ()
examplePipe = do
  x <- request
  y <- request
  liftIO $ print (x + y)

exampleInput :: IO Int
exampleInput = putStr "> " >> readLn

main :: IO ()
main = runPipe exampleInput examplePipe
```

(demo)

Pipes: Requests (2)

```haskell
newtype Pipe a r = Pipe { unPipe :: IO (PipeStep a r) }

data PipeStep a r =
    Pure r
  | Request (a -> Pipe a r)

instance Monad (Pipe a) where
  return x = Pipe $ return (Pure x)
  x >>= f  = Pipe $ do
               xstep <- unPipe x
               case xstep of
                 Request k -> return $ Request (k >=> f)
                 Pure r    -> unPipe (f r)

instance MonadIO (Pipe a) where
  liftIO io = Pipe $ Pure <$> io
```

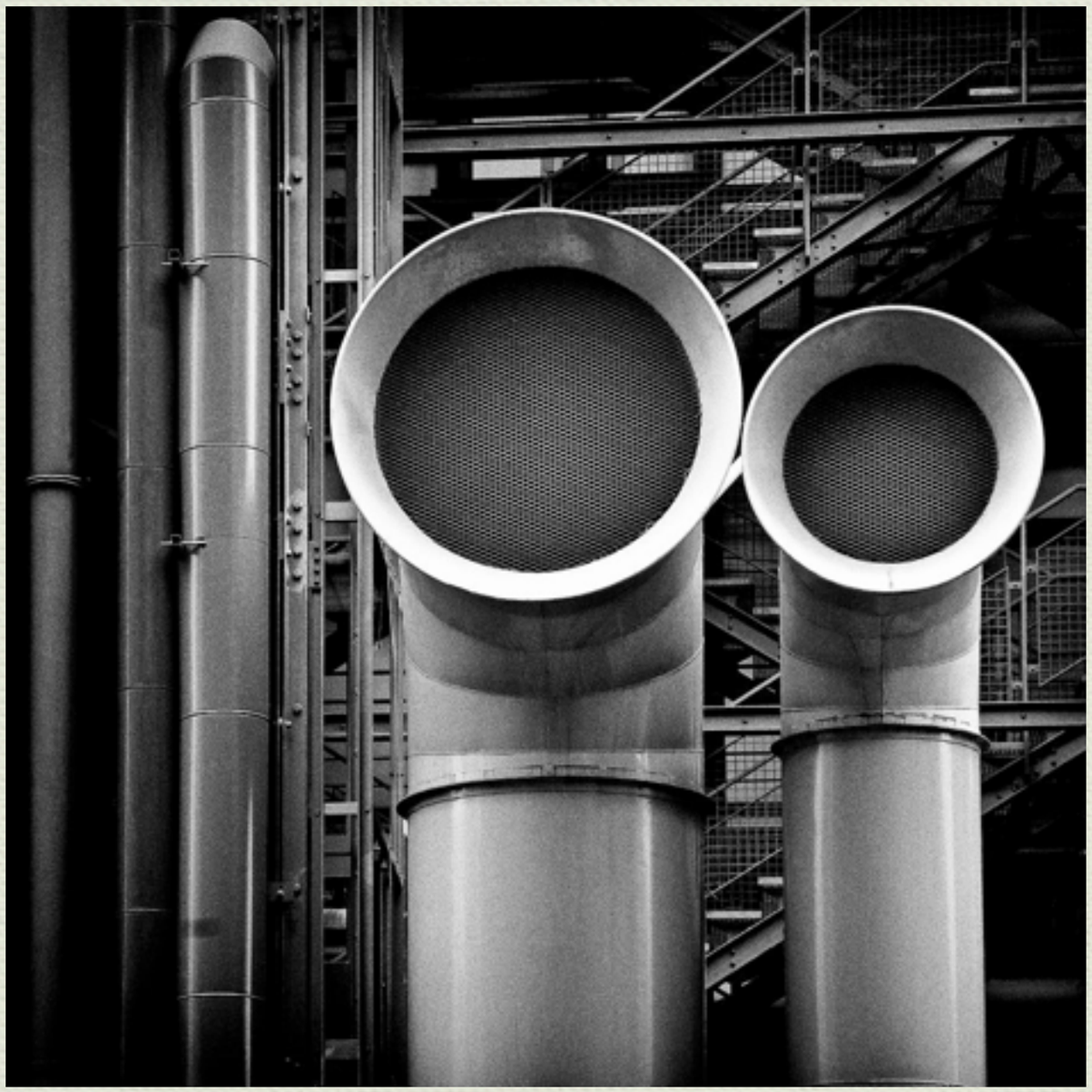
```haskell
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

Pipes: Requests (3)

```
newtype Pipe a r = Pipe { unPipe :: IO (PipeStep a r) }

data PipeStep a r =
    Pure r
  | Request (a -> Pipe a r)

request :: Pipe r r
request = Pipe . return $ Request return

runPipe :: IO a -> Pipe a r -> IO r
runPipe input p = do
  step <- unPipe p
  case step of
    Request k -> input >>= \a -> runPipe input (k a)
    Pure r    -> return r
```

Pipes From Scratch

- Requests
- **Responses**
- Basic pipes
- Finalizers
- Exceptions
- Asynchronous exceptions
- Leftovers



Pipes: Responses (1)

```
examplePipe :: Pipe Int ()
examplePipe = do
  respond 2
  liftIO $ putStr " Hello "
  respond 3
  liftIO $ putStr " world\n"

exampleOutput :: Int -> IO ()
exampleOutput n = replicateM_ n $ putChar '.'

main :: IO ()
main = runPipe exampleOutput examplePipe
```

(demo)

Pipes: Responses (2)

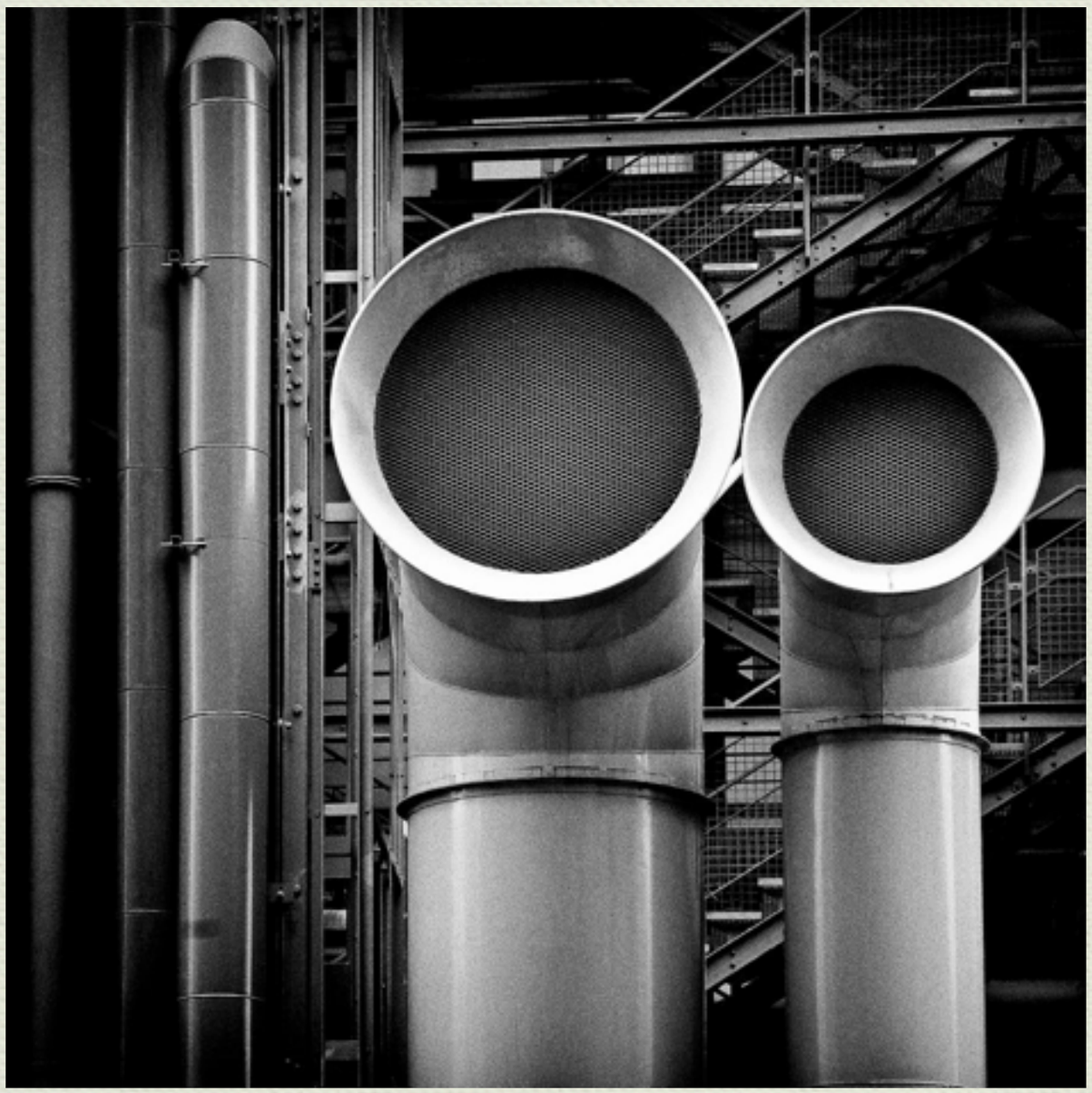
```haskell
newtype Pipe b r = Pipe { unPipe :: IO (PipeStep b r) }

data PipeStep b r =
    Pure r
  | Respond b (Pipe b r)

instance Monad (Pipe b) where
  return x = Pipe $ return (Pure x)
  x >>= f  = Pipe $ do
               xstep <- unPipe x
               case xstep of
                 Respond b k -> return $ Respond b (k >>= f)
                 Pure r      -> unPipe (f r)

instance MonadIO (Pipe a) where
  -- as before
```

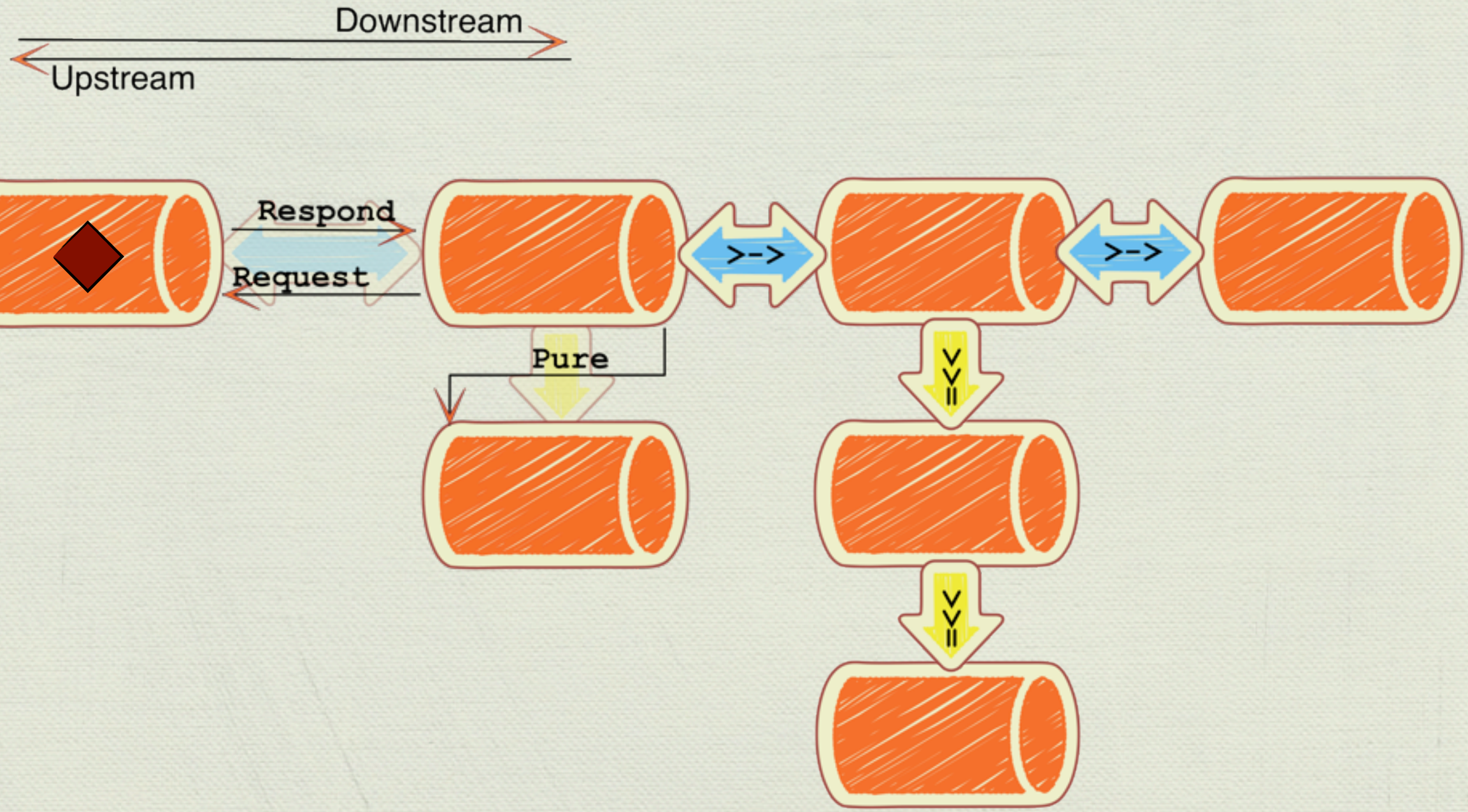
Pipes: Responses (3)

```haskell
newtype Pipe b r = Pipe { unPipe :: IO (PipeStep b r) }

data PipeStep b r =
    Pure r
  | Respond b (Pipe b r)

respond :: b -> Pipe b ()
respond b = Pipe . return $ Respond b (return ())

runPipe :: (b -> IO ()) -> Pipe b r -> IO r
runPipe output p = do
  step <- unPipe p
  case step of
    Respond b k -> output b >> runPipe output k
    Pure r      -> return r
```

Pipes From Scratch

- Requests
- Responses
- **Basic pipes**
- Finalizers
- Exceptions
- Asynchronous exceptions
- Leftovers




Downstream
Upstream
Respond
Request
>-
>-
Pure

Basic Pipes Example (1)

```
prompter :: Pipe a Int r
prompter = do
  liftIO $ putStrLn "Hi!"
  forever $ do
    i <- liftIO $ putStr "> " >> readLn
    respond i

printer :: Pipe Int b r
printer = forever $ do
  i <- request
  liftIO $ print i

main1 :: IO ()
main1 = runPipe (prompter >-> printer)
```

(demo)

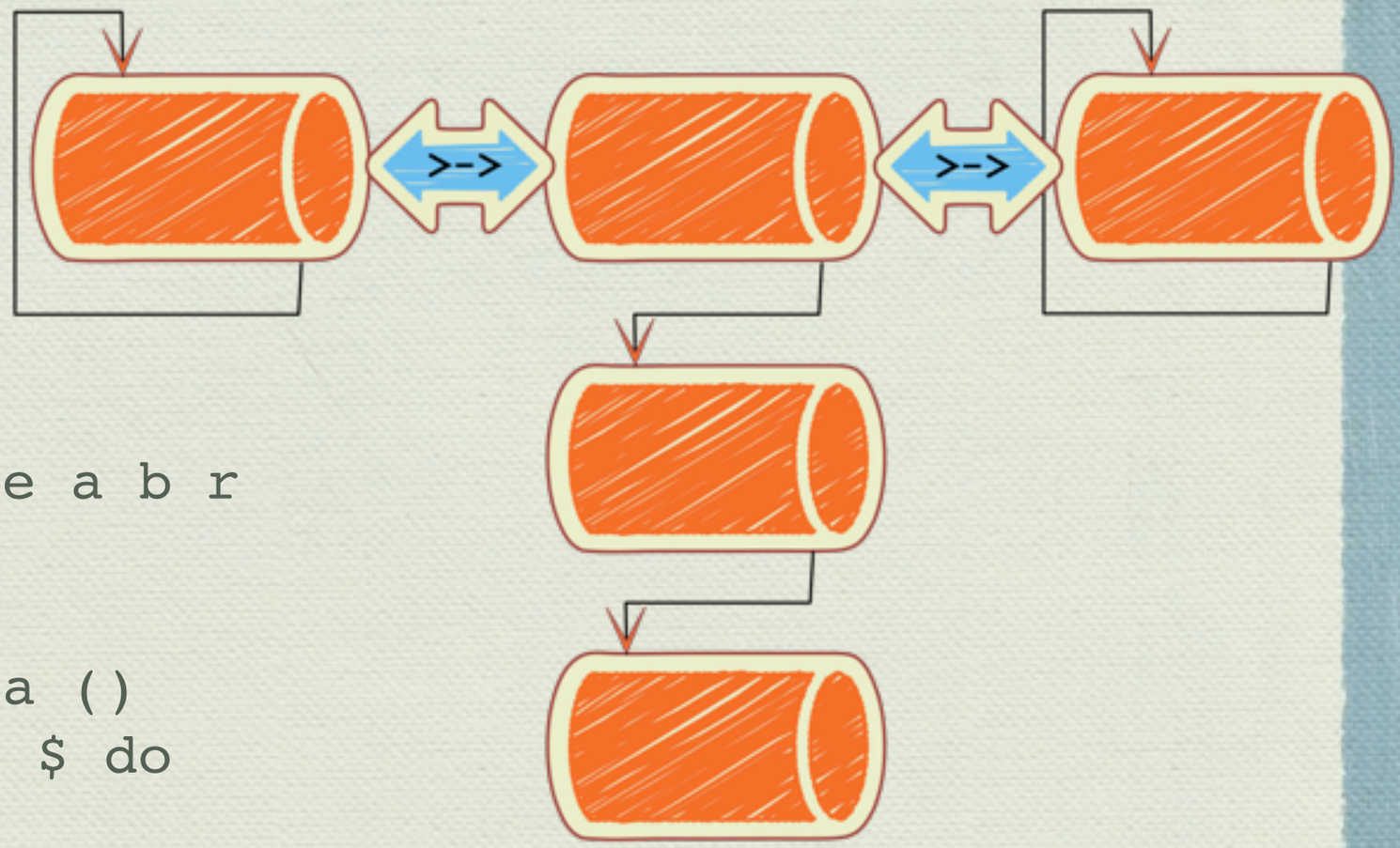
Basic Pipes Example (2)

```
prompter :: Pipe a Int r
prompter = -- as before

printer :: Pipe Int b r
printer = -- as before

mapPipe :: (a -> b) -> Pipe a b r
mapPipe f = forever $ do
  a <- request
  respond (f a)

main2 :: IO ()
main2 = runPipe (prompter >-> mapPipe (* 2) >-> printer)
```

(demo)

Basic Pipes Example (3)

```
prompter :: Pipe a Int r
prompter = -- as before

printer :: Pipe Int b r
printer = -- as before

mapPipe :: (a -> b) -> Pipe a b r
mapPipe f = -- as before

takePipe :: Int -> Pipe a a ()
takePipe n = replicateM_ n $ do
  a <- request
  respond a

main3 :: IO ()
main3 = runPipe (prompter >-> takePipe 3 >-> printer)
```



(demo)

Basic Pipes

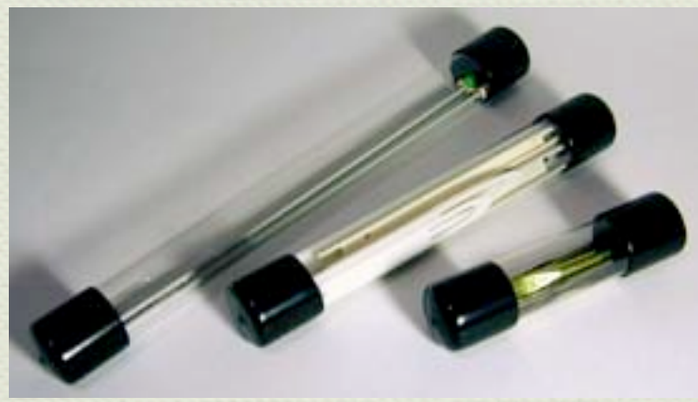
```haskell
newtype Pipe a b r = Pipe { unPipe :: IO (PipeStep a b r) }

data PipeStep a b r =
    Pure r
  | Request (a -> Pipe a b r)
  | Respond b (Pipe a b r)

instance Monad (Pipe a b) where
  -- as before

instance MonadIO (Pipe a b) where
  -- as before

respond :: b -> Pipe a b ()
respond b = -- as before

request :: Pipe r b r
request = -- as before
```

Composition

```
(>->) :: forall a b c r. Pipe a b r -> Pipe b c r -> Pipe a c r
(>->) = goRight
  where
    goRight :: Pipe a b r -> Pipe b c r -> Pipe a c r
    goRight p q = Pipe $ do
      qstep <- unPipe q
      case qstep of
        Respond b k -> return $ Respond b (goRight p k)
        Pure r      -> return $ Pure r
        Request k   -> unPipe $ goLeft p k

  goLeft :: Pipe a b r -> (b -> Pipe b c r) -> Pipe a c r
  goLeft p q = Pipe $ do
    pstep <- unPipe p
    case pstep of
      Request k   -> return $ Request (\a -> goLeft (k a) q)
      Pure r      -> return $ Pure r
      Respond b k -> unPipe $ goRight k (q b)
```

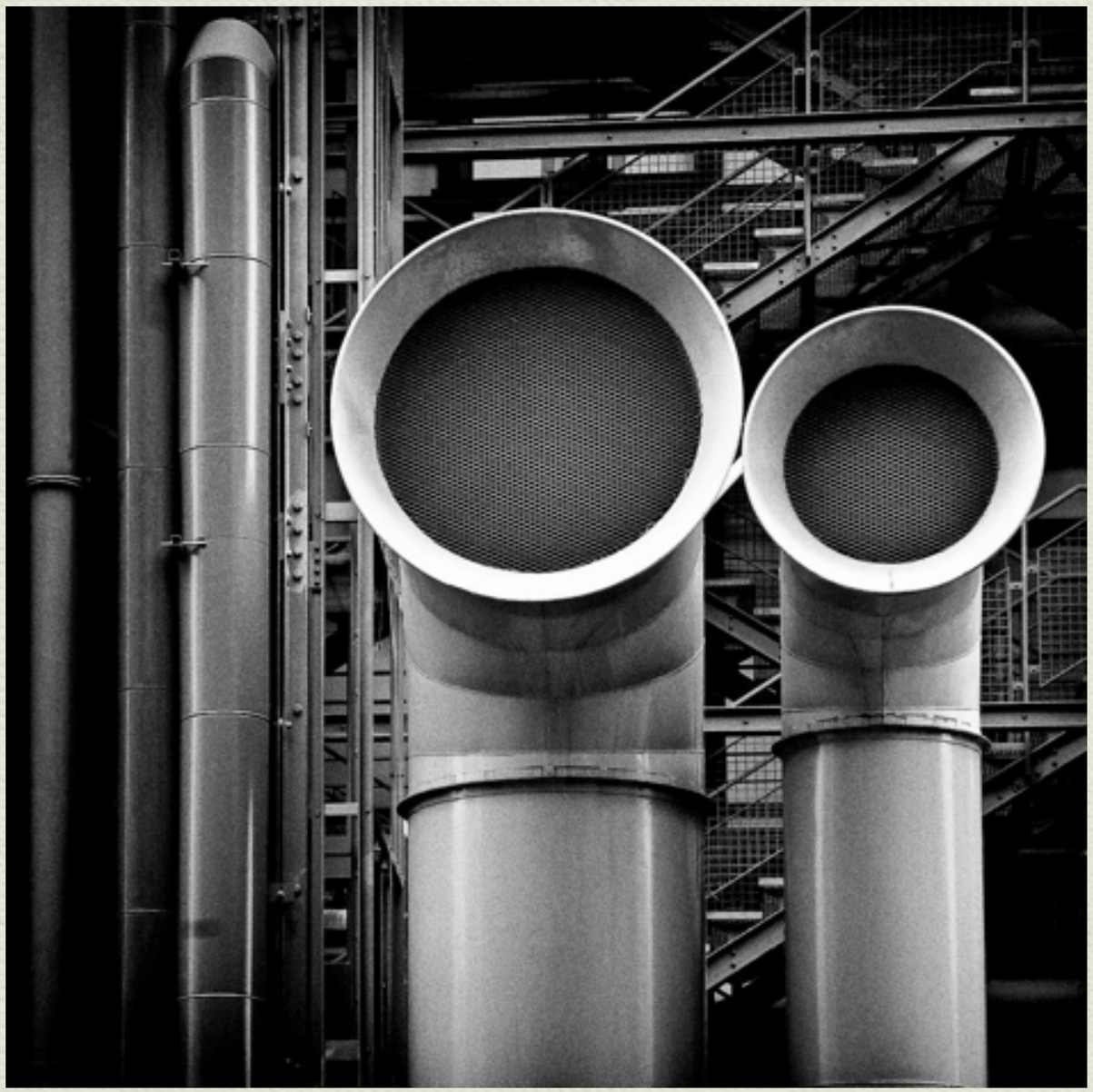
Running a pipe

```
newtype Pipe a b r = Pipe { unPipe :: IO (PipeStep a b r) }

data PipeStep a b r =
    Pure r
  | Request (a -> Pipe a b r)
  | Respond b (Pipe a b r)

runPipe :: Pipe () Void r -> IO r
runPipe p = do
  step <- unPipe p
  case step of
    Pure r      -> return r
    Request k   -> runPipe (k ())
    Respond b _ -> absurd b
```



Pipes From Scratch

- Requests
- Responses
- Basic pipes
- **Finalizers**
- Exceptions
- Asynchronous exceptions
- Leftovers



Using Finalizers

```
prompter :: Pipe a Int r
prompter = finallyP (putStrLn "</prompter>") $ do
  liftIO $ putStrLn "<prompter>"
  forever $ do
    i <- liftIO $ putStr "> " >> readLn
    respond i

printer :: Pipe Int b r
printer = finallyP (putStrLn "</printer>") $ do
  liftIO $ putStrLn "<printer>"
  forever $ do
    i <- request
    liftIO $ print i
```

Finalizer Examples

```
main1 :: IO ()
main1 = runPipe (prompter >-> takePipe 3 >-> printer)

main2 :: IO ()
main2 = runPipe (     forever (prompter >-> takePipe 3)
                  >-> printer)

main3 :: IO ()
main3 = runPipe (     forever (prompter >-> takePipe 3)
                  >-> takePipe 3
                  >-> printer)
```

(demo)

Pipes: Finalizers (1)

```
newtype Pipe a b r = Pipe { unPipe :: IO (PipeStep a b r) }

type Finalizer = IO ()

data PipeStep a b r =
    Pure (Either SomeException r)
  | Request Finalizer (a -> Pipe a b r)
  | Respond Finalizer b (Pipe a b r)

instance Monad (Pipe a b) where
  return x = Pipe $ return (Pure x)
  x >>= f  = Pipe $ do
    xstep <- unPipe x
    case xstep of
      Request z k   -> return $ Request z (k >=> f)
      Respond z b k -> return $ Respond z b (k >>= f)
      Pure r        -> unPipe (f r)
```

In the style of conduit, but conduit has an asymmetry between downstream and upstream termination

Pipes: Finalizers (2)

```
(>->) :: forall a b c r. Pipe a b r -> Pipe b c r -> Pipe a c r
(>->) = goRight (return ())
 where
  goRight :: Finalizer -> Pipe a b r -> Pipe b c r -> Pipe a c r
  goRight z p q = Pipe $ do
   qstep <- unPipe q
   case qstep of
    Respond z' b k -> return $ Respond (z >> z') b (goRight z p k)
    Pure r         -> z >> return (Pure r)
    Request z' k   -> unPipe $ goLeft z' p k

  goLeft :: Finalizer -> Pipe a b r -> (b -> Pipe b c r) -> Pipe a c r
  goLeft z p q = Pipe $ do
   pstep <- unPipe p
   case pstep of
    Request z' k   -> return $ Request (z >> z') (\a -> goLeft z (k a) q)
    Pure r         -> z >> return (Pure r)
    Respond z' b k -> unPipe $ goRight z' k (q b)
```

Upstream finalizer

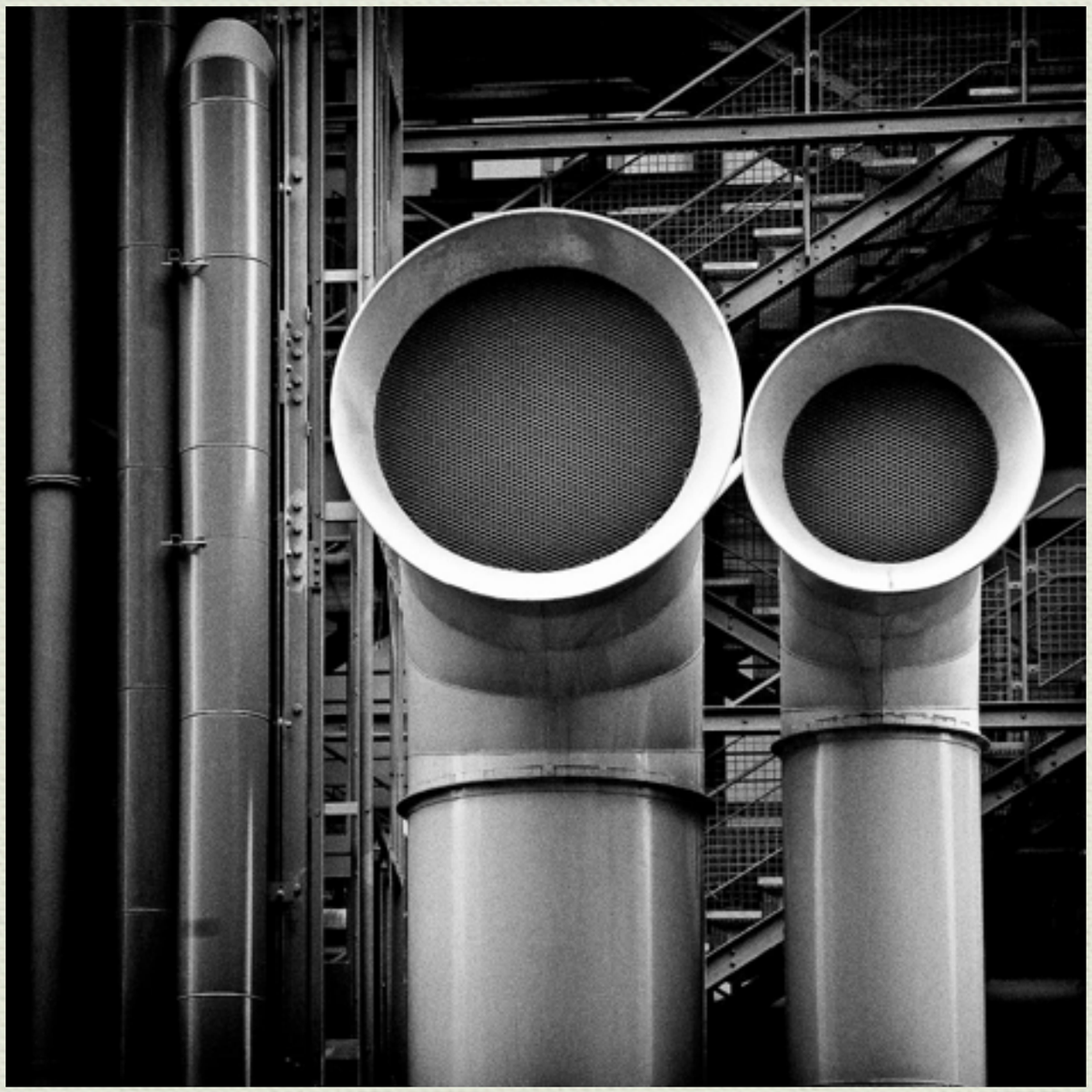
Downstream finalizer

Pipes: Finalizers (3)

```
finallyP :: Finalizer -> Pipe a b r -> Pipe a b r
finallyP z p = Pipe $ do
  step <- unPipe p
  case step of
    Request z' k   -> return $ Request (z' >> z) (\a -> finallyP z (k a))
    Respond z' b k -> return $ Respond (z' >> z) b (finallyP z k)
    Pure r         -> z >> return (Pure r)
```

Pipes From Scratch

- Requests
- Responses
- Basic pipes
- Finalizers
- **Exceptions**
- Asynchronous exceptions
- Leftovers



Example Exceptions (1)

```
exampleHandler2 :: AsyncException -> Pipe a b ()
exampleHandler2 _e =
  liftIO . putStrLn $ "How dare you interrupt me?"

main3 :: IO ()
main3 = runPipe (   (    forever (prompter >-> takePipe 3)
                      `catchP`
                        exampleHandler2
                    )
                  >-> printer)
```

(demo)

Exceptions (1)

```
newtype Pipe a b r = Pipe { unPipe :: IO (PipeStep a b r) }

type Finalizer = IO ()

data PipeStep a b r =
    Pure (Either SomeException r)
  | Request Finalizer (a -> Pipe a b r)
  | Respond Finalizer b (Pipe a b r)

instance Monad (Pipe a b) where
  return x = Pipe $ return (Pure (Right x))
  x >>= f  = Pipe $ do
    xstep <- unPipe x
    case xstep of
      Request z k    -> return $ Request z (k >=> f)
      Respond z b k  -> return $ Respond z b (k >>= f)
      Pure (Right r) -> unPipe (f r)
      Pure (Left e)  -> return $ Pure (Left e)
```

Exceptions (2)

```haskell
instance MonadIO (Pipe a b) where
  liftIO io = Pipe $ Pure <$> try io

catchP :: Exception e
       => Pipe a b r -> (e -> Pipe a b r) -> Pipe a b r
catchP p h = Pipe $ do
  step <- unPipe p
  case step of
    Pure (Left e)  -> case fromException e of
                        Just e' -> unPipe $ h e'
                        Nothing -> return $ Pure (Left e)
    Pure (Right r) -> return $ Pure (Right r)
    Request z k    -> return $ Request z (\a -> catchP (k a) h)
    Respond z b k  -> return $ Respond z b (catchP k h)

-- respond, request, (>->), finallyP as before
```

Example Exceptions (2)

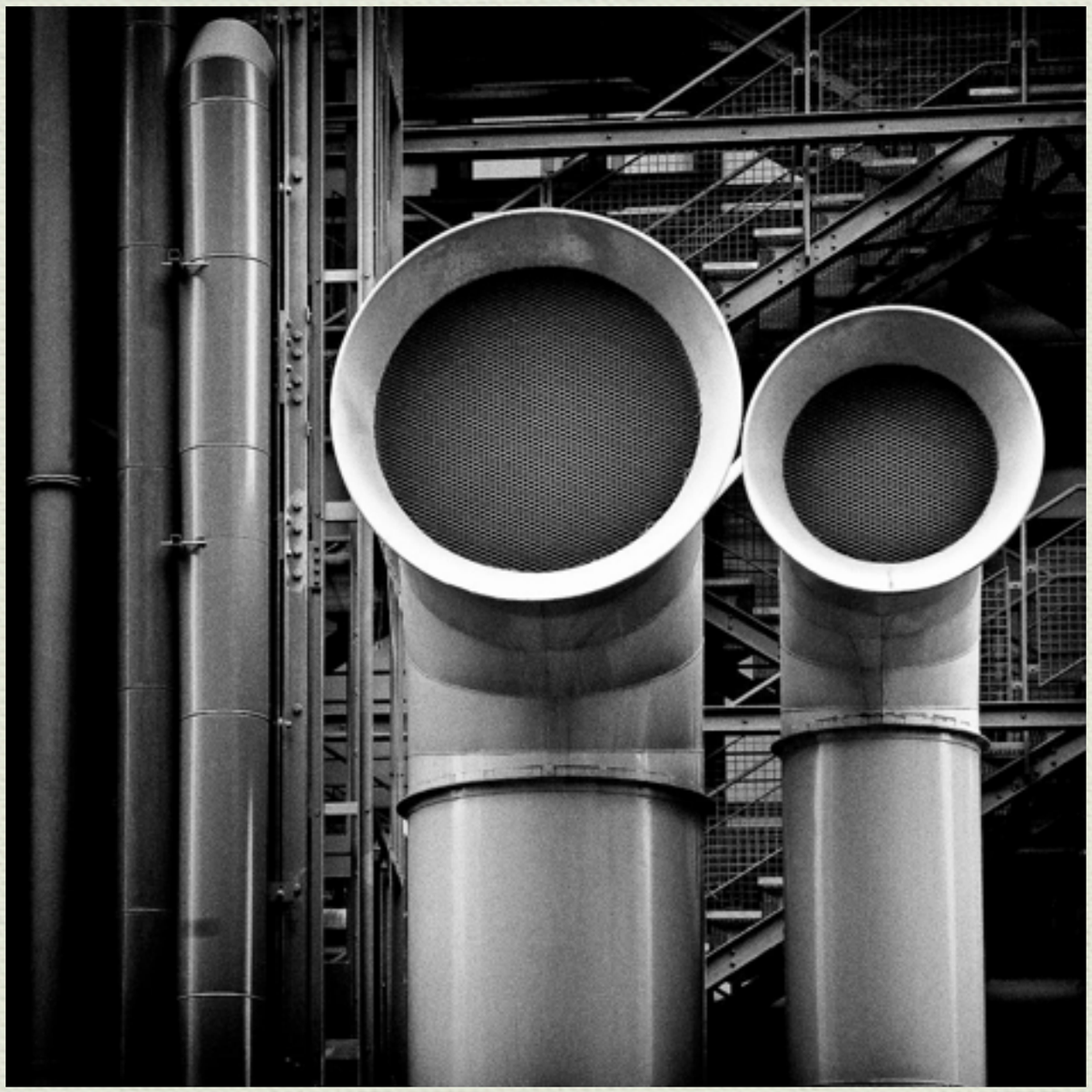
```
fib :: Int -> Pipe a b Int
fib 0 = return 1
fib 1 = return 1
fib n = do
  n_2 <- fib (n - 2)
  n_1 <- fib (n - 1)
  return (n_2 + n_1)

mapPipeM :: (a -> Pipe a b b) -> Pipe a b r
mapPipeM p = forever $ do
  a <- request
  b <- p a
  respond b

main4 :: IO ()
main4 = runPipe (prompter >-> mapPipeM fib >-> printer)
```

Intentionally slow fibonacci

(demo)

Pipes From Scratch

- Requests
- Responses
- Basic pipes
- Finalizers
- Exceptions
- **Asynchronous exceptions**
- Leftovers



Async Exceptions (Local Approach)

```
type Restore = forall a. IO a -> IO a

newtype Pipe a b r = Pipe {
   unPipe :: Restore -> IO (PipeStep a b r)
 }

instance Monad (Pipe a b) where
  return x = Pipe $ \_ -> return (Pure (Right x))
  x >>= f  = Pipe $ \restore -> do
   xstep <- try . restore $ unPipe x restore
   case xstep of
    Left  err                -> return $ Pure (Left err)
    Right (Request z k)      -> return $ Request z (k >=> f)
    Right (Respond z b k)    -> return $ Respond z b (k >>= f)
    Right (Pure (Right r)) -> unPipe (f r) restore
    Right (Pure (Left e))    -> return $ Pure (Left e)

runPipe :: Pipe () Void r -> IO r
runPipe = \p -> mask $ \restore -> ...
```

Pure computation cannot be interrupted

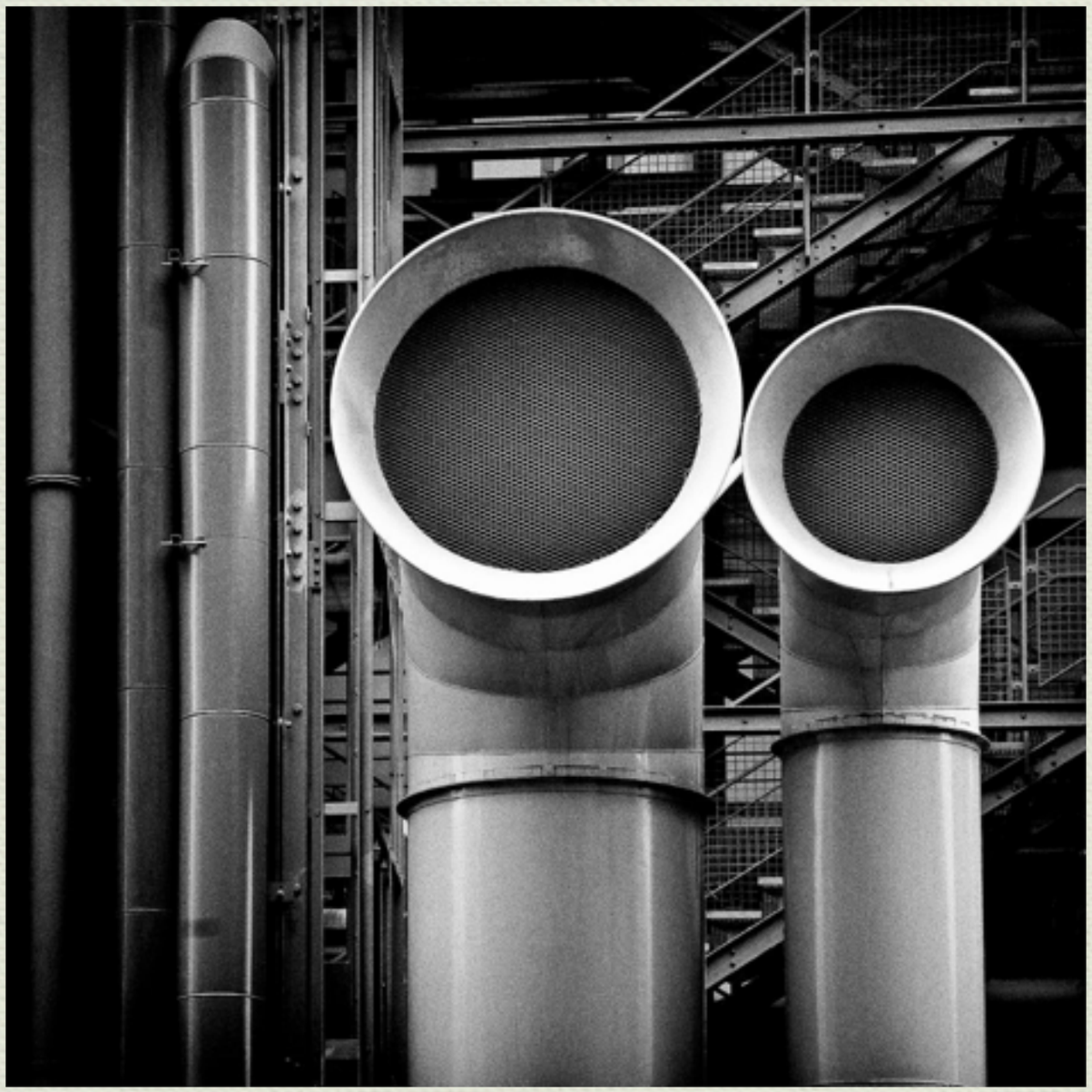
Async Exceptions (Global Approach)

```
newtype Pipe a b r = Pipe
  { unPipe :: ResourceT IO (PipeStep a b r) }

finallyP :: IO () -> Pipe a b r -> Pipe a b r
finallyP z p = Pipe $ register z >>= \key -> unPipe $ go key p
  where
    go :: ReleaseKey -> Pipe a b r -> Pipe a b r
    go key p = ...

runPipe :: Pipe () Void r -> IO r
runPipe = runResourceT . go
  where
    go :: Pipe () Void r -> ResourceT IO r
    go p = ...
```

Async exceptions allowed anywhere, but less clear when exceptions are caught

Pipes From Scratch

- Requests
- Responses
- Basic pipes
- Finalizers
- Exceptions
- Asynchronous exceptions
- **Leftovers**



Leftovers Example

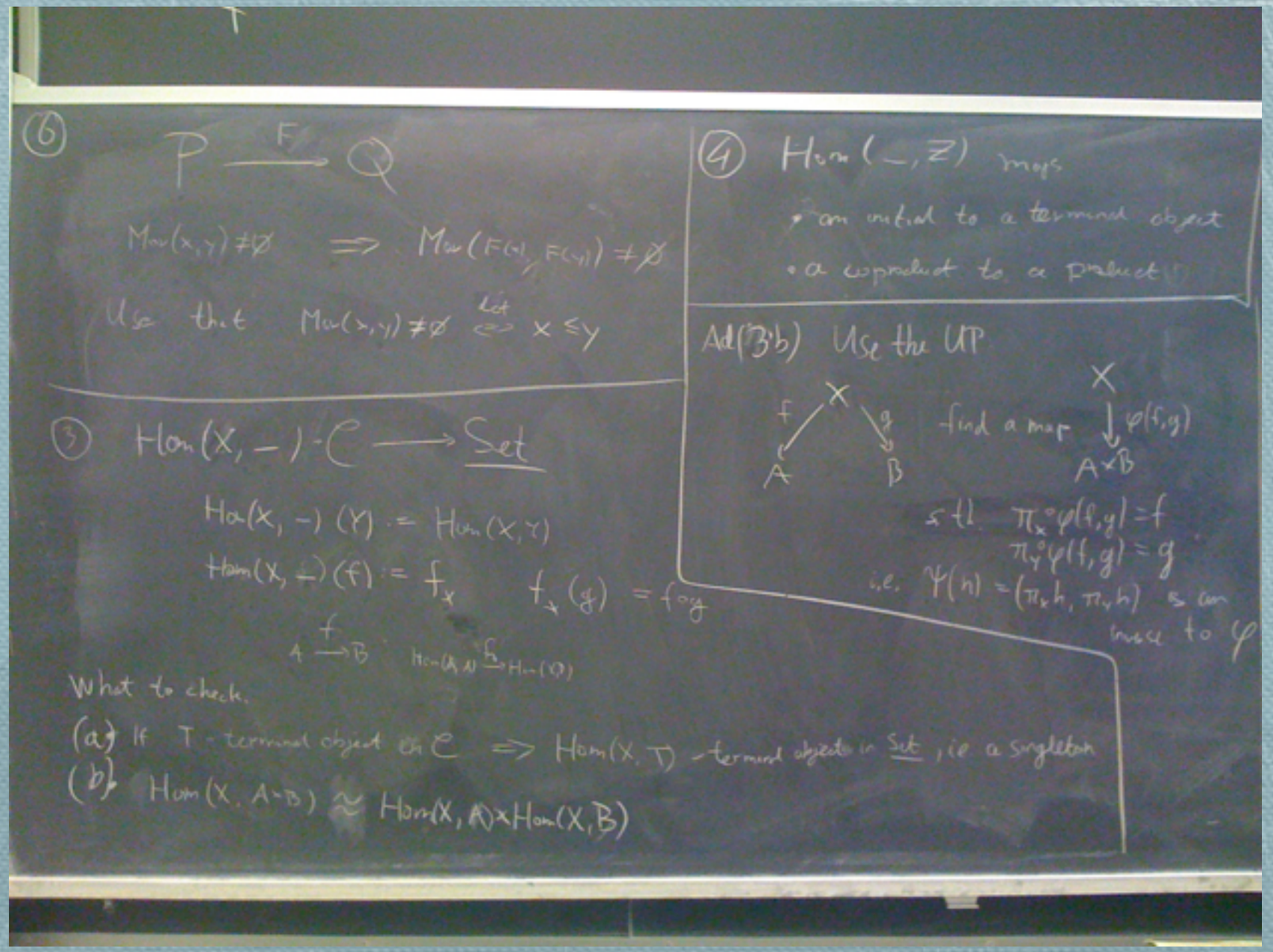
```
dropWhile :: (l -> Bool) -> Pipe l l b ()
dropWhile p = do
  x <- request
  if p x
    then dropWhile p
    else leftover x

idPipe :: Pipe a a a r
idPipe = forever $ request >>= respond

main1 :: IO ()
main1 = runPipe (     prompter
                  >-> (dropWhile even >> idPipe)
                  >-> printer)
```

(demo)

Leftovers (conduit style)

```haskell
newtype Pipe l a b r = Pipe { unPipe :: IO (PipeStep l a b r) }

data PipeStep l a b r =
  -- ...
  | Leftover l (Pipe l a b r)

(>->) :: Pipe l a b r -> Pipe b b c r -> Pipe l a c r
(>->) = goRight
  where
    goRight p q = Pipe $ do
      qstep <- unPipe q
      case qstep of
        -- ...
        Leftover l k -> unPipe $ goRight (respond l >> p) k

    goLeft p q = Pipe $ do
      pstep <- unPipe p
      case pstep of
        -- ...
        Leftover l k -> return $ Leftover l (goLeft k q)
```

breaks the laws :(



Generalizations

Monad Transformer

```haskell
newtype Pipe a b m r = Pipe { unPipe :: m (PipeStep a b m r) }

-- PipeStep as before

instance Monad m => Monad (Pipe a b m) where -- as before

instance MonadTrans (Pipe a b) where
  lift p = Pipe $ Pure `liftM` p

instance MonadIO m => MonadIO (Pipe a b m) where
  liftIO = lift . liftIO

runPipe :: Monad m => Pipe () Void m r -> m r
runPipe p = -- as before
```

Pipe Transformers (1)

```
class Pipe p where
  -- Monad
  returnP :: r -> p a b r
  bindP   :: p a b r -> (r -> p a b r') -> p a b r'
  -- MonadIO
  liftIOP :: IO r -> p a b r
  -- Pipe specific
  request :: p r b r
  respond :: b -> p a b ()
  (>->)   :: p a b r -> p b c r -> p a c r

instance Pipe p => Monad (p a b) where
  return = returnP
  (>>=)  = bindP

instance Pipe p => MonadIO (p a b) where
  liftIO = liftIOP
```

Pipe Transformers (2)

```haskell
newtype IdPipe a b r = Pipe { unPipe :: IO (PipeStep a b r) }

data PipeStep a b r =
    Pure r
  | Request (a -> IdPipe a b r)
  | Respond b (IdPipe a b r)

instance Pipe IdPipe where -- as before
```

Pipe Transformers (3)

```
newtype MaybeP p a b r = MaybeP { unMaybeP :: p a b (Maybe r) }

instance Pipe p => Pipe (MaybeP p) where
  returnP x   = MaybeP $ returnP (Just x)
  x `bindP` f = MaybeP $ do ma <- unMaybeP x
                            case ma of
                              Nothing -> return Nothing
                              Just a  -> unMaybeP (f a)

  liftIOP io = MaybeP $ Just `liftM` liftIOP io

  respond b = MaybeP $ Just `liftM` respond b
  request   = MaybeP $ Just `liftM` request
  p >-> q   = MaybeP $ unMaybeP p >-> unMaybeP q

abort :: Pipe p => MaybeP p a b r
abort = MaybeP $ return Nothing
```

Pipe Transformers (4)

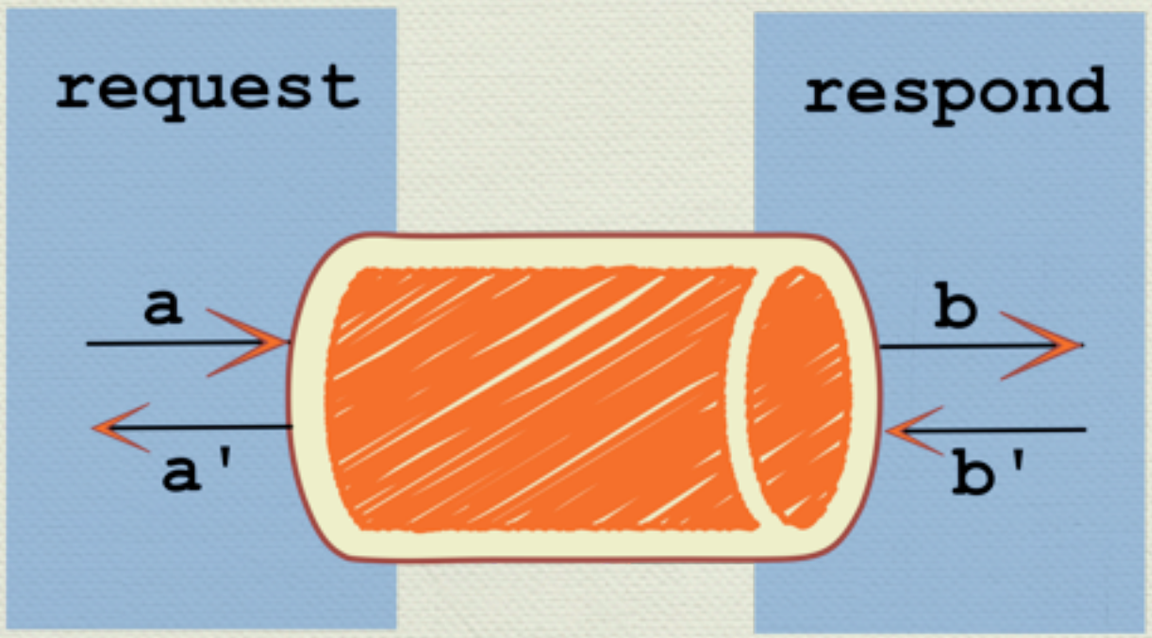
```
prompter :: Pipe p => p a Int r
prompter = -- implementation exactly as before

printer :: Pipe p => p Int b r
printer = -- implementation exactly as before

abortIf :: Pipe p => (a -> Bool) -> MaybeP p a a r
abortIf p = forever $ do
  a <- request
  if p a then abort
         else respond a

main2 :: IO (Maybe ())
main2 = runIdPipe . unMaybeP $ prompter
                           >-> abortIf even
                           >-> printer
```

(demo)

Bidirectional Pipes (1)

```haskell
newtype Pipe a' a b' b r = Pipe
  { unPipe :: IO (PipeStep a' a b' b r) }

data PipeStep a' a b' b r =
    Pure r
  | Request a' (a  -> Pipe a' a b' b r)
  | Respond b  (b' -> Pipe a' a b' b r)

-- Monad and MonadIO instance, as well as respond and request,
-- are as you would expect
```

Bidirectional pipes are known are also called **proxies**



Bidirectional Pipes (2)

```
doubleServer :: Int -> Pipe a' a Int Int r
doubleServer req = do
  req' <- respond (req * 2)
  doubleServer req'

client :: [Int] -> () -> Pipe Int a b' a ()
client is () = mapM_ aux is
  where
    aux i = do
      resp <- request i
      respond resp

main4 :: IO ()
main4 = runPipe (doubleServer >-> client [1, 2, 3] >-> printer)
```

(demo)

Bidirectional Pipes (3)

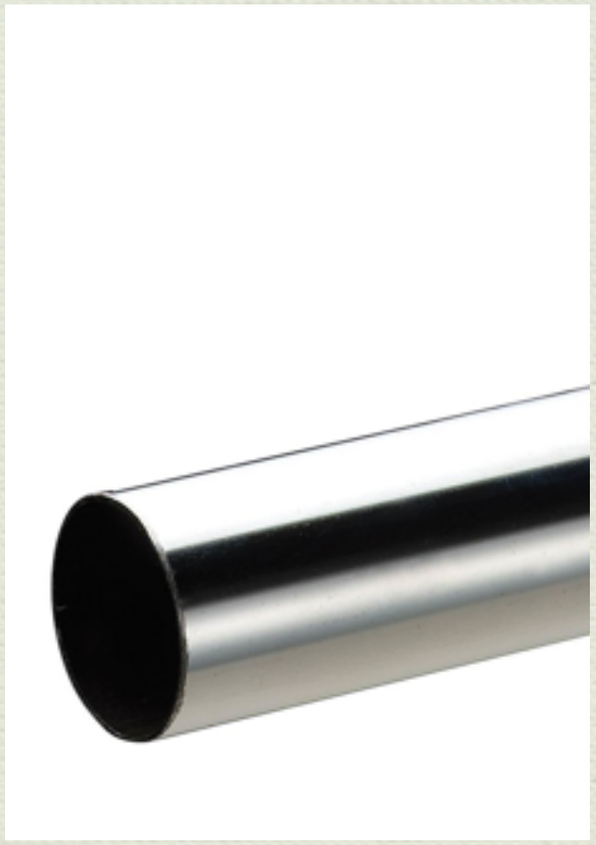
```
(>->) :: (b' -> Pipe a' a b' b r)
      -> (c' -> Pipe b' b c' c r)
      -> (c' -> Pipe a' a c' c r)
p >-> q = \c' -> p ->> q c'

(->>) :: (b' -> Pipe a' a b' b r)
      ->        Pipe b' b c' c r
      ->        Pipe a' a c' c r
p ->> q = Pipe $ -- similar to goRight, before

(>>~) ::        Pipe a' a b' b r
      -> (b -> Pipe b' b c' c r)
      ->        Pipe a' a c' c r
p >>~ q = Pipe $ -- similar to goLeft, before
```

Symmetry between
`->>` (`goRight`)
and
`>>~` (`goLeft`)
more evident

Alternatives

Terminology

source
producer
enumerator



pipe
conduit
enumeratee

sink
consumer
iteratee

“enumerator”

```
data Step a m b
   = Continue (Stream a -> Iteratee a m b)
   | Error Exc.SomeException
   | Yield b (Stream a)

newtype Iteratee a m b = Iteratee
   { runIteratee :: m (Step a m b) }
```

“iterator” (un-CPS-transformed)

```
data Step a m b
   = Continue (Stream a -> Iteratee a m b)
              (Maybe SomeException)
   | Yield b (Stream a)

newtype Iteratee a m b = Iteratee
   { runIteratee :: m (Step a m b) }
```

“iterIO”

```
data IterR t m a
  = IterF !(Iter t m a)
  | IterM !(m (IterR t m a))
  | IterC !(CtlArg t m a)
  | Done a (Chunk t)
  | Fail !IterFail !(Maybe a) !(Maybe (Chunk t))

newtype Iter t m a = Iter
  { runIter :: Chunk t -> IterR t m a }
```

Finally: Shout with Pipes

```
shout :: FilePath -> FilePath -> IO ()
shout inPath outPath =
  runSafeIO . runProxy . runEitherK $
          readFileS inPath
    >-> mapD (map toUpper)
    >-> writeFileD outPath
```

References: Pipes

- http://hackage.haskell.org/package/pipes
- http://hackage.haskell.org/package/pipes-safe
- http://hackage.haskell.org/package/pipes-parse
- http://hackage.haskell.org/package/mmorph
- Blog posts at http://www.haskellforall.com

References: Conduit

- http://hackage.haskell.org/package/conduit
- http://hackage.haskell.org/package/resourcet
- https://www.fpcomplete.com/user/snoyberg/library-documentation
- https://github.com/snoyberg/conduit/wiki/Dealing-with-monad-transformers
- http://www.yesodweb.com/blog ; http://www.yesodweb.com/book/conduits
- http://unknownparallel.wordpress.com/2012/07/24/pipes-to-conduits-part-0-combining-functors/ and following blog posts
- http://www.yesodweb.com/blog/2012/01/conduit-versus-enumerator

References: Others

- http://hackage.haskell.org/package/iteratee
- http://hackage.haskell.org/package/enumerator
- http://hackage.haskell.org/package/iterIO
- http://www.yesodweb.com/blog/2012/01/conduit-versus-enumerator
- http://www.yesodweb.com/blog/2010/09/enumerators-tutorial-part-1 ; http://www.yesodweb.com/blog/2010/10/enumerators-tutorial-part-2 ; http://www.yesodweb.com/blog/2010/10/enumerators-tutorial-part-3
- http://okmij.org/ftp/Streams.html

References: IO-Streams

- http://hackage.haskell.org/package/io-streams
- http://snapframework.com/blog/2013/03/05/announcing-io-streams

References: Other

- "Iteratee I/O", http://www.haskell.org/haskellwiki/Iteratee_I/O
- "Coroutine Pipelines", Issue 19 of The Monad Reader
- "Another way of looking at traversal", Real World Haskell, http://book.realworldhaskell.org/read/io-case-study-a-library-for-searching-the-filesystem.html#find.fold
- "High Performance Monads" (nice explanation of monads in CPS form), http://blog.ezyang.com/2010/09/high-performance-monads/
- "Simulating Quantified Class Constraints", Valery Trifonov, Haskell '03 (explains "returnP" and co)

unsafePerformIO

:: IO a -> a

```
ex1 :: Int
ex1 = unsafePerformIO $ randomRIO (1, 6)

ex1_mainA = print (even (ex1 + ex1))
ex1_mainB = print (even (2 * ex1))
```

```
ex2 :: Int
{-# INLINE ex2 #-}
ex2 = unsafePerformIO $ randomRIO (1, 6)

ex2_mainA = print (even (ex2 + ex2))
ex2_mainB = print (even (2 * ex2))
```

```
ex3 :: Int -> Int
ex3 i = unsafePerformIO $ randomRIO (1, 6)

ex3_mainA = print (even (ex3 1 + ex3 1))
ex3_mainB = print (even (2 * ex3 1))
```

```
stdout :: Handle
{-# NOINLINE stdout #-}
stdout = unsafePerformIO $ do
   setBinaryMode FD.stdout
   enc <- getLocaleEncoding
   mkHandle FD.stdout "<stdout>" WriteHandle True (Just enc)
                nativeNewlineMode{-translate newlines-}
                (Just stdHandleFinalizer) Nothing
```