$$
\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}
$$

$$
\begin{aligned}
\eta\, a \gg\!= f &\equiv f\, a \\
m \gg\!= \eta &\equiv m \\
(m \gg\!= f) \gg\!= g &\equiv m \gg\!= (\lambda x \to f\, x \gg\!= g)
\end{aligned}
$$

```
f = do
  putStr “Name: “
  n <- getLine
  putStrLn “Hello “ ++ n
```

Monads

13 Feb 2012

ACM

chris.vanhorne@gmail.com

This presentation contains slides entirely of code.

If you begin to feel nauseous, please exit immediately to the rear of the room.

- Functional programming matters. John Hughes wrote a famous paper about the topic.

- Multicore now. Threads won't save you.

- Small and understandable functions with clear composition semantics are the future. "Software alchemy" will be a metric soon.

- Lazy evaluation is a good default for exploratory programming: combinators; infinite structures; etc.

Software Alchemy

“Write a dot-product method for you data type...”

“Write a dot-product method for you data type...”

```
public static double dot(Vec a, Vec b) {
  double dp = 0;
  int min = (int) Math.min(a.len(), a.len());
  for (int i = 0; i < min; i++)
    dp += a.get(i)*b.get(i);
  return dp;
}
```

"Write a dot-product method for you data type..."

```
public static double dot(Vec a, Vec b) {
  double dp = 0;
  int min = (int) Math.min(a.len(), a.len());
  for (int i = 0; i < min; i++)
    dp += a.get(i)*b.get(i);
  return dp;
}
```

- Seem redundant?
- Seem error-prone?
- Did you catch the bug?

“Write a dot-product method for you data type...”

“Write a dot-product method for you data type...”

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
foldl   :: (a -> b -> a) -> a -> [b] -> a

dot xs ys = sum (join xs ys)
  where
    sum  = foldl1 (+)
    join = zipWith (*)
```

“Write a dot-product method for you data type...”

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
foldl   :: (a -> b -> a) -> a -> [b] -> a

dot xs ys = sum (join xs ys)
  where
    sum  = foldl1 (+)
    join = zipWith (*)
```

- Where did we take the *length* of our list?
- Do you trust this version? Why?
- There’s still a bug. Phantom types can give us type-level errors for mixing dimensions.

- Composable and provable functions help turn complex software into composable and provable software.

- *“A map. I know what map does,”* as opposed to, *“hmm, for-loop, not sure if looping or ...”*

- This power comes from the purity of our functions. Did the *dot* function launch missiles. How do you know?

Pure-lazy Conundrum

People say, I'll use a pure-lazy IO now they have a

```
main () =
  putStr “Enter your name: “;
  name <- getLine;
  putStrLn “Hello, “ ++ name;
```

* Assume a pure-lazy Haskell-like language executing line-based statements.

```
main () =
  putStr “Enter your name: “;
  name <- getLine;
  putStrLn “Hello, “ ++ name;
```

* Assume a pure-lazy Haskell-like language executing line-based statements.

- This should execute exactly how you expect. It doesn’t.
- When did we ask for the value of *name*? What do you think it prints now?
- When did we ask for the value of line one?

$$f\,(g(x)) \equiv (f \circ g)\,x$$

- We need to force an evaluation order.
- Function composition forces order!
- Problem solved. The world rejoices.

- Great. We can do something about IO.
- What about logging? Is this IO?
- Environments?
- State?
- ???

Wadler Interpreter

http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf

```haskell
data Term = Const Int | Div Term Term

eval :: Term -> Int
eval (Const a) = a
eval (Div x y) = eval x `div` eval y
```

```haskell
data Term = Const Int | Div Term Term

eval :: Term -> Int
eval (Const a) = a
eval (Div x y) = eval x `div` eval y


> eval (Div (Const 9) (Const 3))
3
```

```haskell
data Term = Const Int | Div Term Term

eval :: Term -> Int
eval (Const a) = a
eval (Div x y) = eval x `div` eval y


> eval (Div (Const 9) (Const 3))
3
```

- Great. Handle division-by-zero errors. Hmmmm..
- What's an error? I know. I'll make a data type...

```haskell
data M a = Raise String | Return a
data Term = Const Int | Div Term Term

eval :: Term -> M Int
eval (Const a) = Return a
eval (Div x y) =
  case eval x of
    Raise e -> Raise e  -- propagate
    Return a ->
      case eval y of
        Raise e -> Raise e  -- ugh..
        Return b ->
          if b == 0
            then Raise “divide by zero”
            else Return (a `div` b)
```

Well. That was horrible. What about a simple, pure, logger? Should be easy.

```
type M a = (String, a)
data Term = Const Int | Div Term Term

eval :: Term -> M Int
eval (Const a) = (“Const”, a)
eval (Div x y) =
  let (u, a) = eval x in
  let (v, b) = eval y in
  (“Div “ ++ u ++ v , a `div` b)


> eval (Div (Const 9) (Const 3))
(“Div Const Const”, 3)
```

They look kind of similar...

```
type M a = (String, a)
data Term = Const Int | Div Term Term

eval :: Term -> M Int
eval (Const a) = ("Const", a)
eval (Div x y) =
  let (u, a) = eval x in
  let (v, b) = eval y in
  ("Div " ++ u ++ v , a `div` b)
```

datatype M a = *computation structure*

```
data Term = Const Int | Div Term Term

eval :: Term -> M Int
eval (Const a) = ("Const", a)
eval (Div x y) =
  let (u, a) = eval x in
  let (v, b) = eval y in
  ("Div " ++ u ++ v , a `div` b)
```

```
datatype M a = computation structure
data Term = Const Int | Div Term Term

eval :: Term -> M a
eval (Const a) = ("Const", a)
eval (Div x y) =
  let (u, a) = eval x in
  let (v, b) = eval y in
  ("Div " ++ u ++ v , a `div` b)
```

```
datatype M a = computation structure
data Term = Const Int | Div Term Term

eval :: Term -> M a
eval (Const a) = make a trivial “M a”
eval (Div x y) =
  let (u, a) = eval x in
  let (v, b) = eval y in
  (“Div “ ++ u ++ v , a `div` b)
```

```
datatype M a = computation structure
data Term = Const Int | Div Term Term

eval :: Term -> M a
eval (Const a) = make a trivial “M a”
eval (Div x y) =
  eval x and pass on result to..
    eval y and pass on result to..
      construct M (a `div` b).
```

Monads

- We need a generic way to get "into" our monad. *Exception* used *Return*, *Logger* constructed a tuple of `(String, Int)`.

- In our *eval* function, the act of passing on a *result* had specific hidden logic to the particular monad we were working in.

- Let's break down each of these requirements and see if we can create a general framework for <u>computations.</u>

“Return” into our monad

- We have: `data M a = M a`.
- We want to take `a` into `M`.
- Thus, `a -> M a`.

"Return" into our monad

- We have: `data M a = M a`.
- We want to take `a` into `M`.
- Thus, `a -> M a`.

```
return :: a -> M a
return a = M a
```

“Bind” pure function into our monadic chain

- Given a monad, `M a`, apply a pure function, `f`, and give me the result.

“Bind” pure function into our monadic chain

- Given a monad, `M a`, apply a pure function, `f`, and give me the result.

```
bind :: M a -> (a -> b) -> M b
```

“Evaluate and continue” in our monad.

- Given a monad, `M a`, apply a pure function, `f`, and give me the result.

```
bind :: M a -> (a -> b) -> M b
```

Where did `M b` come from?

"Evaluate and continue" in our monad.

- Given a monad, `M a`, apply a pure function, `f`, and give me the result.

```
bind :: M a -> (a -> b) -> M b
```

Extract 'a' from monad, feed into pure function.

“Evaluate and continue” in our monad.

- Given a monad, `M a`, apply a pure function, `f`, and give me the result.

```
bind :: M a -> (a -> b) -> M b
```

Apply function to pure value of computation, return back into a computation. Type is still wrong...

“Evaluate and continue” in our monad.

- Given a monad, `M a`, apply a <u>neat</u> function, `f`, and give me the result.

```
bind :: M a -> (a -> M b) -> M b
```

There we go!

- How do we apply pure functions to computations if we need a result that is a computation... :-(

- Easy: how do we get into the monad?

```
id k = k

\k -> return . id
```

```
id k = k

\k -> return . id
```

```
id k = k

\k -> return . id

                  a -> M a
```

All Together Now.

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

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


data Maybe a = Nothing | Just a
```

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

```
data Maybe a = Nothing | Just a
```

* This is not a monad. We make it a monad by satisfying the Monad interface. It’s easy!

- What's the most trivial `Maybe` computation?

- If we have a computation which is `Nothing`, what kind of computation should we get if we apply a function `f` to it?

- If we have `Just x`, what kind of computation should we get after applying a function `f` to it?

```
return x = Just x

Nothing  >>= f = Nothing

(Just x) >>= f = f x
```

```haskell
instance Monad Maybe where
  return          = Just
  Nothing  >>= f = Nothing
  (Just a) >>= f = f a
```

```haskell
Just 42 >>= return . id             == Just 42
Nothing >>= return . id             == Nothing
Just 42 >>= return . id >>= Nothing == Nothing
```

```haskell
instance Monad Maybe where
  return              = Just
  Nothing  >>= f = Nothing
  (Just a) >>= f = f a
```

```haskell
Just 42 >>= return . id              == Just 42
Nothing >>= return . id              == Nothing
Just 42 >>= return . id >>= Nothing == Nothing
```

Wait.. Nothing combined with Something equals Nothing? That's kind of like our division-by-zero error propagation..

Safe Division... M-M-M-Monad

We can make these now.

```haskell
eval :: Term -> M Int
eval (Const k) = return k
eval (Div x y) =
  eval x >>= \m ->
  eval y >>= \n ->
  return $ safeDiv m n

safeDiv :: Int -> Int -> Maybe Int
safeDiv m 0 = Nothing
safeDiv m n = Just $ m `div` n
```

```
eval :: Term -> M Int
eval (Const k) = return k
eval (Div x y) = do
  m <- eval x
  n <- eval y
  return $ safeDiv m n

safeDiv :: Int -> Int -> Maybe Int
safeDiv m 0 = Nothing
safeDiv m n = Just $ m `div` n
```

Looks like line-sequenced imperative program!

That's great. What if I want those side-effect things like printing to the console?

That's great. What if I want those side-effect things like printing to the console?

What can we do in our monadic bind function?

That's great. What if I want those side-effect things like printing to the console?

What can we do in our monadic bind function?

(Almost) anything we want.

That's great. What if I want those side-effect things like printing to the console?

What can we do in our monadic bind function?

(Almost) anything we want.

Our *Logger* monad looks like a good candidate. He had lots of magical logging "computational facets" during bind.

End Game

```haskell
newtype Logger m a = Logger { runLogger :: (a, m) }
```

```haskell
newtype Logger m a = Logger { runLogger :: (a, m) }
```

```haskell
newtype Logger m a = Logger { runLogger :: (a, m) }
```

Type of Logger.
Implementation will
restrict this.

```
newtype Logger m a = Logger { runLogger :: (a, m) }
```

Type of Logger. Implementation will restrict this.

This is left for the user of a monad to fill in their type.

```haskell
newtype Logger m a = Logger { runLogger :: (a, m) }
```

Type of Logger. Implementation will restrict this.

This is left for the user of a monad to fill in their type.

```
data Maybe a = Just a | Nothing
```

```
newtype Logger m a = Logger { runLogger :: (a, m) }
```

Type of Logger. Implementation will restrict this.

This is left for the user of a monad to fill in their type.

```
data Maybe a = Just a | Nothing
```

```
newtype Logger m a = Logger { runLogger :: (a, m) }
```

Type of Logger. Implementation will restrict this.

This is left for the user of a monad to fill in their type.

```
data Maybe a = Just a | Nothing
```

```
newtype Logger m a = Logger { runLogger :: (a, m) }
```

Type of Logger. Implementation will restrict this.

This is left for the user of a monad to fill in their type.

```
data Maybe a = Just a | Nothing
```

```haskell
newtype Logger m a = Logger { runLogger :: (a, m) }
```

Type of Logger. Implementation will restrict this.

runLogger :: Logger m a -> (a, m)
Gets us “out” of a Logger.

```haskell
instance String s => Monad (Logger s) where
  return a = Logger (a, “”)
```

```
instance String s => Monad (Logger s) where
  return a = Logger (a, “”)

  (Logger (a, log)) >>= f =
```

```
instance String s => Monad (Logger s) where
  return a = Logger (a, “”)

  (Logger (a, log)) >>= f =
    let (Logger (b, log’)) = f a
    in   ...
```

```
* Hint Hint Hint *

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

f :: a -> Logger s b
```

```
instance String s => Monad (Logger s) where
  return a = Logger (a, “”)

  (Logger (a, log)) >>= f =
    let (Logger (b, log’)) = f a
    in   Logger (b, log ++ log’)
```

Computational detail that’s hidden from the users of the monad. Ooooh, shiny!

```
instance String s => Monad (Logger s) where
  return a = Logger (a, “”)

  (Logger (a, log)) >>= f =
    let (Logger (b, log’)) = f a
    in   Logger (b, log ++ log’)



let expr = eval (Div (Const 9) (Const 3))
in snd $ runLogger expr
```

Loggers only on Strings? How about anything which may be appended?

Now you're thinking like a future programmer.

This is ...

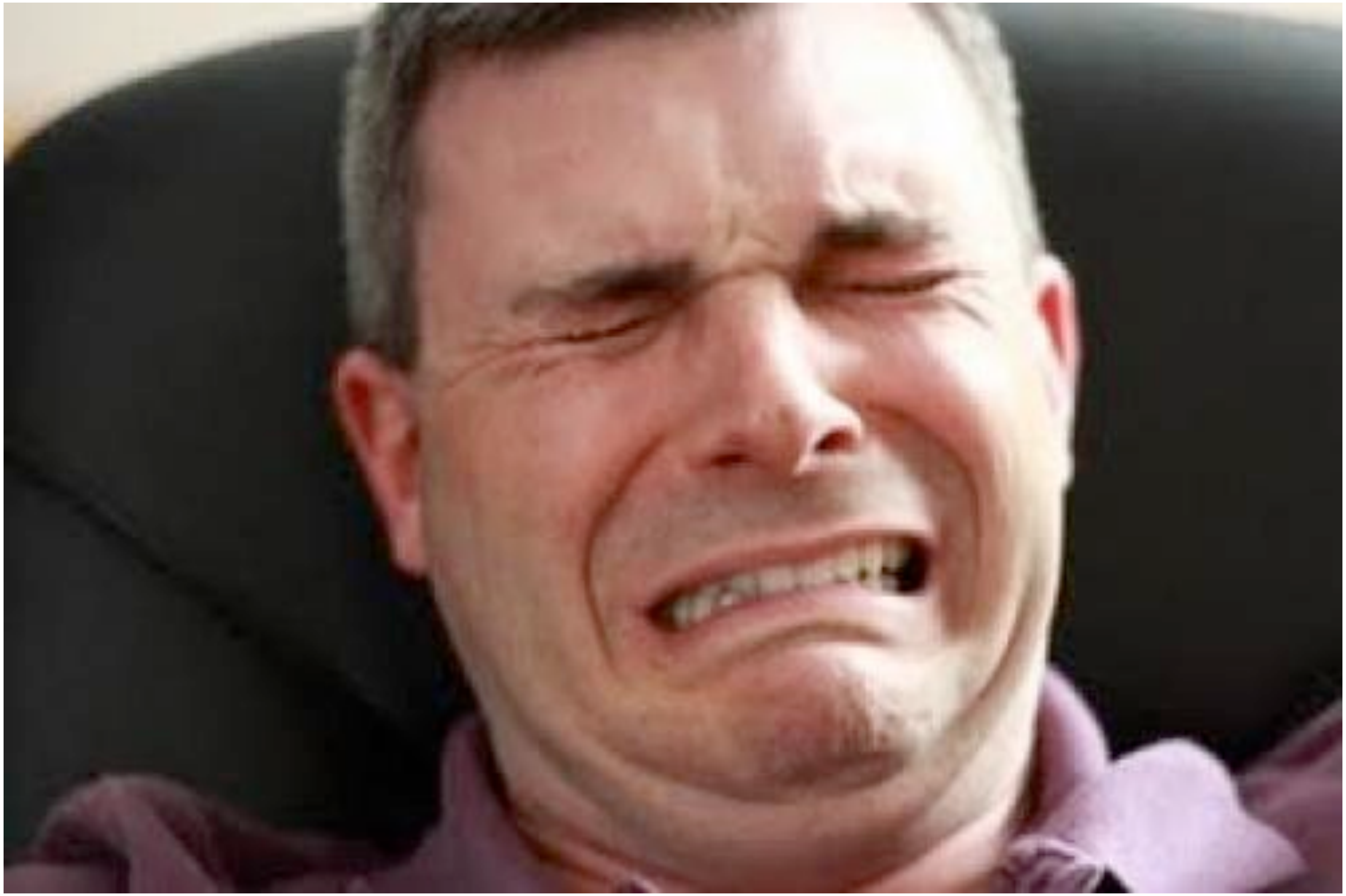
W-W-W-RITER

```haskell
newtype Writer w a = Writer { runWriter :: (a, w) }

instance Monoid m => Monad (Writer m) where
  return a = Writer (a, mempty)

  m >>= f = Writer $ let (a, w)  = runWriter m
                         (b, w’) = runWriter (f a)
                     in  (b, w `mappend` w’)

tell :: Monoid m => Writer m ()
tell m = Writer ((), m)
```

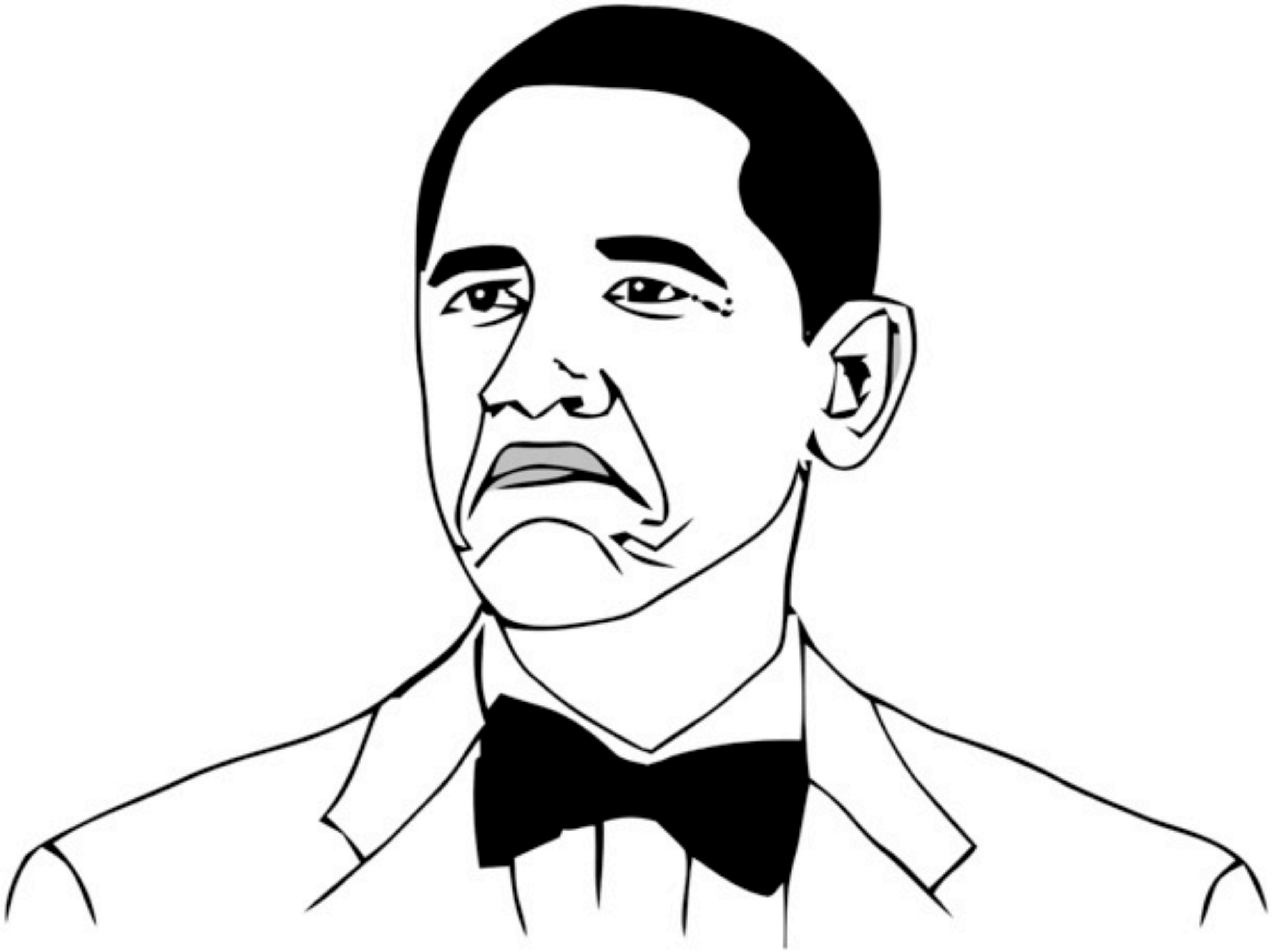
```haskell
eval :: Term -> Writer [String] Int
eval (Const k) = return k
eval (Div x y) = do
  a <- eval x
  b <- eval y
  tell [“Awesome logging”]
  return $ a `div` b
```

```
> runWriter $ eval (Div (Const 9) (Const 3))
(3,["Awesome logging"])
```

We did it together.



MONADS ARE



NOT BAD

What's the meaning of it all?

- Congratulations. Go forth, live on a higher plane of computational and data abstraction.

- Explore the State monad, it's easy after doing Writer. Then look at running programs backwards with the Reverse State monad. It's still easy!

- There are three laws to being a monad. Basically, be left- and right-associative and commute.

Yaron Minsky @yminsky Close

Programming systems with complex error handling in OCaml makes me wonder how the rest of the world manages without algebraic datatypes...

Manuel Chakravarty @TacticalGrace 4 Feb

@bos31337 Alchemy programmers still rule the industry, but non-academic blogs discussing things like product-sum types goes in the right dir

In reply to Bryan O'Sullivan

Edward Kmett @kmett 20 Jul

jQuery is decidedly not a Functor, nor is it an Applicative, or a Monad. reddit.com/r/haskell/comm...

Chris Lonnen @Lonnen 3 Feb

Some people see a problem and think "I know, I'll use Java!" Now they have a ProblemFactory.

Yaron Minsky @yminsky 17 Ma

I'll never understand how the software world tolerates languages whose datatypes can say "and" but not "or".