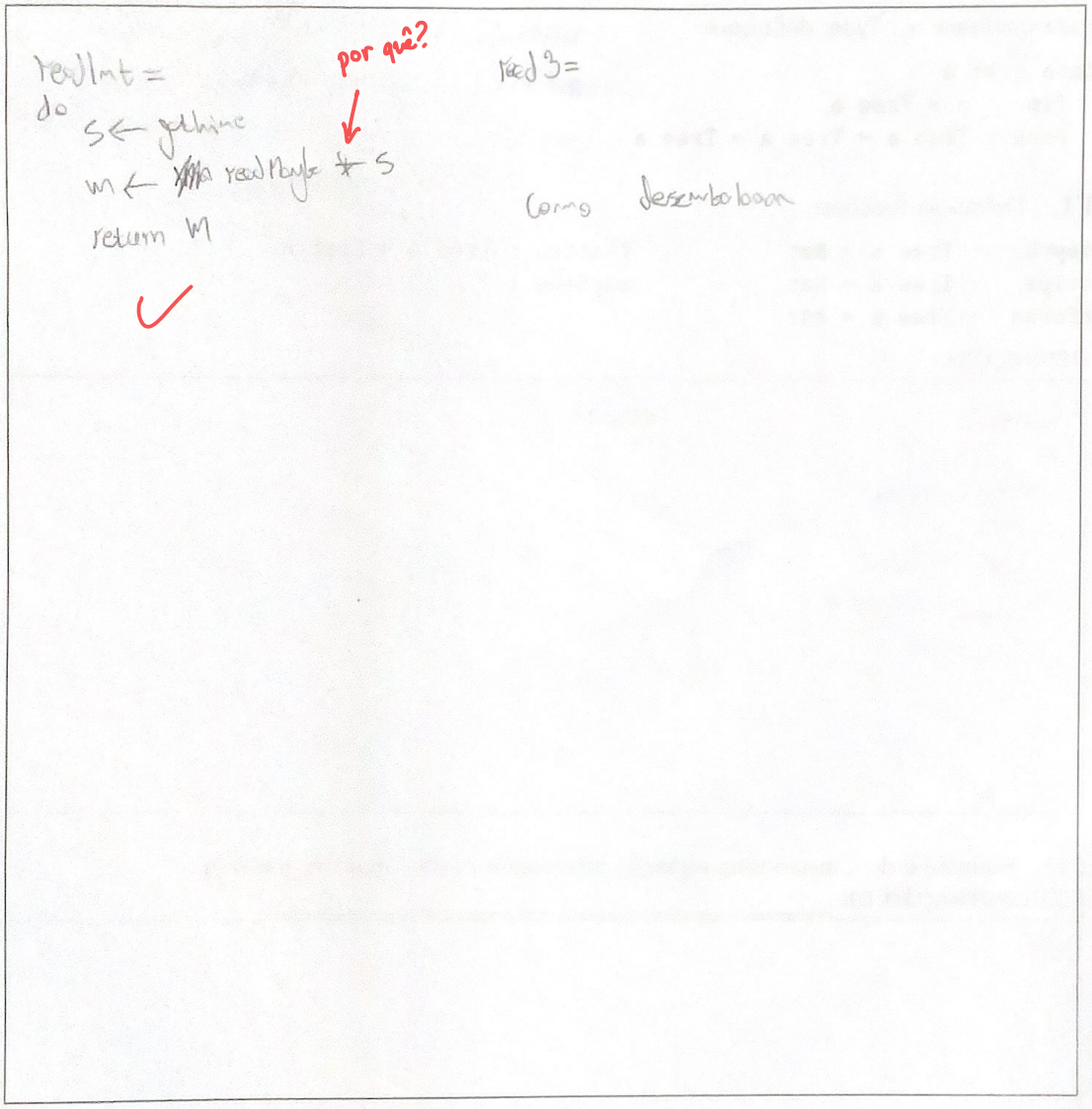

readInt =
do
s ← getLine
m ← readMaybe s
return m
por quê?
read3 =
como desembolsar

Não consegui acompanhar suficientemente bem a unidade 3 :(Pretendo estudar nas férias para alinhar com o que foi passado de Functors/ Applicatives/Monads. Lamento.

Dê o devido carinho ao assunto e tu vai amar! ☺

readInt : IO(M Int)
readInt = do

~~Fmap : (γ → α) → (α → β) → (γ → β) → (α → β)~~

~~fmap fmap~~

$$\underset{(\to\gamma)}{Fmap} : (\alpha \to \beta) \to (\gamma \to \gamma)$$

Fmap



Rafaela

LEMMATA

```
wrap = return

readInt =
  do str <- getLine
     mInt <- readMaybe str
     wrap mInt

read3 =
  do mx <- readInt
     case mx of
       Nothing -> wrap Nothing
       Just x -> my <- readInt
                 case my of
                   Nothing -> wrap Nothing
                   Just y -> mz <- readInt
                             case mz of
                               Nothing -> wrap Nothing
                               Just z -> wrap $ Just (x,y,z)
```



Functor para (δ →):

fmap = (∘)

Preferi definir a (=<<):

$(\alpha \to m\beta) \xrightarrow{fmap} (m\alpha \to mm\beta) \xrightarrow{(join \circ)} (m\alpha \to m\beta)$

Pra definir a join:

join = (=<<) id

Pra definir a fmap:

$(\alpha \to \beta) \xrightarrow{(pure \circ)} (\alpha \to m\beta) \xrightarrow{(=<<)} (m\alpha \to m\beta)$

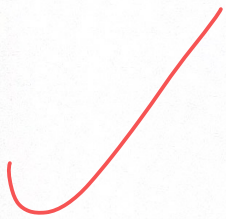
unit = pure ()

```
sequenceAL :: Applicative f => [f a] -> f [a]
sequenceAL []     = pure []
sequenceAL (f:fs) = pure (:) <*> f <*> sequenceAL fs
```

pure (:) <*> f

```
readInt :: IO (Maybe Int)
readInt = do s <- getLine
             pure (readMaybe s)

read3 :: IO (Maybe (Int, Int, Int))
read3 = do x <- readInt
           case x of
             Nothing -> pure Nothing
             Just m -> do y <- readInt
                          case y of
                            Nothing -> pure Nothing
                            Just m -> do z <- readInt
                                         case z of
                                           Nothing -> pure Nothing
```



Lemmata

f =<< mx = join (fmap f mx)

fmap f mx = (pure . f) =<< mx

join = (=<< id)

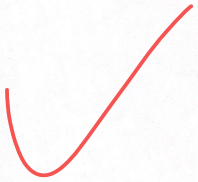
unit = pure ()

pure x = ?

ma ** mb = (pure (,)) <*> ma <*> mb

```
readInt :: IO (Maybe Int)
readInt =
  do
    s <- getLine
    mx = readMaybe s
    pure mx
```

```
read3 :: IO (Maybe (Int, Int, Int))
read3 =
  do
    mx <- readInt
    mx
      Nothing => pure Nothing
      Just x =>
        my <- readInt
        my
          Nothing -> pure Nothing
          Just y ->
            mz <- readInt
            mz
              Nothing -> pure Nothing
              Just z ->
                pure (Just (x, y, z))
```



LEMMATA

instance Functor $\delta \to$. where ①

fmap f g = g ∘ f

id = id

fmap, join ⊢ (>>=) ②

(>>=) mx f = join (fmap f mx)

(>>=), pure ⊢ join ③

join mma = mma >>= id

>>=, pure ⊢ fmap ④

fmap f mx = mx >>= (pure ∘ f)

pure, ⊛ ⊢ unit ⑤

unit = pure ()

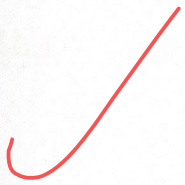
pure, ⊛ ⊢ (*, *) ⑥

(* *) na nb = (⊛) ((⊛) (pure (,)) na) nb

infix!

```
readInt :: IO (m Int)
readInt = do
  s ← getLine
  mx ← readMaybe s
  Pure mx


read3 :: IO( M ( Int, Int, Int))
read3 = do
   mx ← readInt
   case mx of
      nothing ↝ Pure nothing
      Just x ↝ do
         my ← readInt
         case my of
            nothing ↝ Pure nothing
            Just y ↝ do
               mz ← readInt
               case mz of
                  nothing ↝ Pure nothing
                  Just z ↝ (Pure ∘ Just) (x, y, z)
```



Unit, (**) ⊢ Pure, ⊗

Pure a = fmap (\() -> a) Unit

f ⊗ a = fmap (uncurry ($)) (f ** a)

Pure, ⊗ ⊢ Unit, (**)

Unit = Pure ()

f ** g = Pure (,) ⊗ f a ⊗ f b

fmap, Join ⊢ (>>=)

ma >>= f = Join (fmap f ma)

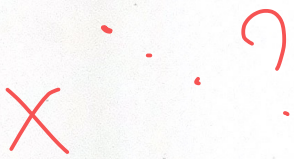
(>>=), Pure ⊢ Join

Join = (>>= id)

readMaybe : String → M α
getLine

```
readInt : IO (M Int)
readInt = do
            x ← getLine
            return $ readMaybe x

read3 : IO (M (Int, Int, Int))
read3 = do
          x ← readInt
          case x of
            Nothing = return
```



unit, (**) ⊣⊢ pure, ⊛

fmap, ϱjoin ⊢ (>>=)

fmap, ϱjoin ⊣ >>= , pure

functo para um dos: (δ →) / (→ α)

exemplo (0) mas não id

sequence Al : App f ⇒ List (f α) → f (List α)

fa ⊛ x =

fmap, ϱjoin ⊢ >>=

ma >>= f = fmap f (ϱjoin ma)

sequence Al : App f ⇒ List (f α) → f (List α)

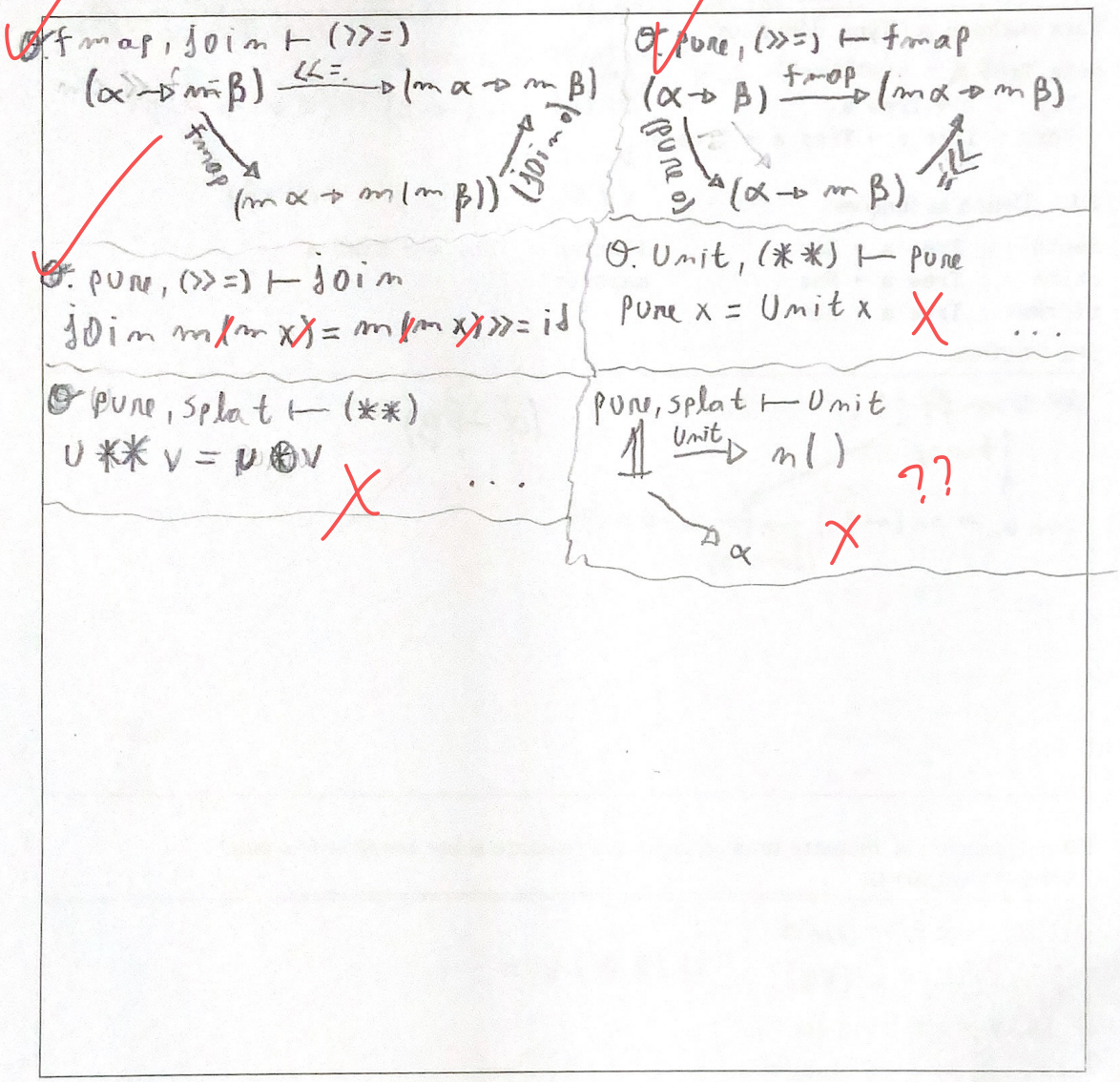
sequence Al [] = pure []

sequence (x : xs) = ~~pure (~~

(**) = ① a ** b =

readInt : Io (MInt)




fmap, join ⊢ (>>=)
(α → m β) <<= (m α → m β)
fmap
(m α → m (m β))
(join ∘ fmap)
pure, (>>=) ⊢ fmap
(α → β) fmap (m α → m β)
pure
(α → m β)
pure, (>>=) ⊢ join
join m (m x) = m (m x) >>= id
Unit, (**) ⊢ pure
pure x = Unit x
X
...
pure, splat ⊢ (**)
u ** v = u ⊛ v
X
...
pure, splat ⊢ Unit
Unit
m ()
α
X
??

```
read3 :: IO (M (Int, Int, Int))
read3 = do
```

```
readInt :: IO (M Int)
readInt = do
  x ← (readMaybe ∘ getLine) x
```

??

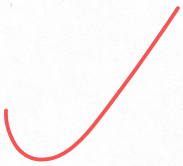
I couldn't do any.



Respostas
LEMMATA

```
readInt : IO (Maybe Int)
readInt =
do
  mi <- readMaybe
  return mi
```

```
read3Int :: IO (Maybe (Int, Int, Int))
read3Int =
do
  x <- readInt
  y <- readInt
  z <- readInt
  case x of
    Nothing -> return Nothing
    Just x' -> case y of
                 Nothing -> return Nothing
                 Just y' -> case z of
                              Nothing -> return Nothing
                              Just z' -> return (x', y', z')
```



Respostas

Lemmata

```
pure = Unit<**>
```

```
Unit = pure ()
```

```
ma >>= mab = join (fmap mab (pure ma))
```

```
readInt : IO (M Int)
readInt = do  x ← getLine
              y ← readMaybe x
              pure y
```

DADOS :
* readMaybe : String → M α
* getLine : IO String

```
read3 : IO (M (Int, Int, Int))
read3 = do  x ← readInt
            y ← readInt
            z ← readInt
```

exatamente
isso é o
que gostaríamos
de escrever
e o que conseguimos
mesmo com monad transformers!



pure , ⊛

⊢ $unit_n : M\ 1$

$unit_n = pure\ ()$

⊢ $(**)_n : n\alpha \times n\beta \to M(\alpha \times \beta)$

$ax\ **\ bx = pure\ (,) \circledast ax \circledast bx$

Functor $(\delta \to)$

$fmap : (\alpha \to \beta) \to (\delta \to \alpha) \to (\delta \to \beta)$

$fmap\ f\ g = f \circ g$

fmap , join

⊢ $(\gg=) : m\ \alpha \to (\alpha \to m\beta) \to m\beta$

$ax \gg= f = join\ (fmap\ f\ ax)$

$(\gg=)$, pure

⊢ $join : m(m\alpha) \to m\alpha$

$join\ axx = axx \gg= id$

⊢ $fmap : (\alpha \to \beta) \to m\ \alpha \to m\ \beta$

$fmap\ f\ ax = ax \gg= (pure \circ f)$

1) readInt :

```
readInt = do str <- getLine
             pure $ readMaybe str
```

2) read3 :

```
read3 = do mx <- readInt
           my <- readInt
           mz <- readInt
           case mx of
             Nothing -> pure $ Nothing
             Just x -> case my of
                         Nothing -> pure $ Nothing
                         Just y -> case mz of
                                     Nothing -> pure $ Nothing
                                     Just z -> pure $ Just (x,y,z)
```



unit, (**) ⊢ pure, <*>

1. pure x = (const x) <*> unit ✗
2. ff <*> fa = uncurry ($) <$> (ff ** fa) ✗

pure, <*> ⊢ unit, (**)

3. unit = pure () ✓
4. (**) = (<*>) . ((,) <$>) ✗

instance Functor (S →) where

5. fmap = (∘) ✓

fmap, join ⊢ (>>=), pure

6. pure x = ✗

(>>=) = (join ∘) . (<$>)

sequenceA [] = pure []

sequenceA (x : xs) = sequenceA xs <$> pure (:) <*> x

Grr...

readInt : IO (M Int) readMaybe : String → M α

getline : IO String

read3 : IO (M (Int, Int, Int))

read3 : ?

```
do
x ← readInt
y ← readInt
z ← readInt
if x = nothing
break
~~else y = nothing~~
~~break~~
if y = nothing
break
if z = nothing
break
-- functiondo (,,) = g
pure g
.
.
.
```

que if é esse?

como define isso?

readInt :

```
do
x ← getline
readMaybe x = y
pure y
```

?

readMaybe :

```
readMaybe "s" = "ss"
readMaybe s = if readMaybe s
.
.
.
readMaybe [] = error
readMaybe (x:xs) = x
.
.
.
```

? ?

X

essa parte é necessária

