Indução

lec 12

2024-11-06

0. (+)-ass : $(\forall n)(\forall m)(\forall k)[(n+m)+k = n+(m+k)]$

DEMONS:

Seja n : Nat.
Seja m : Nat.
Seja k : Nat.
Sep em casos k :

CASO 0 :

CASO Sk :

DADOS

n : Nat

m : Nat

k : Nat

ALVO

$(n+m)+k = n+(m+k)$

data Nat

  O : Nat

  S : Nat → Nat

data List α

  Nil : List α

  Cons : α → List α → List α

Θ? $(\forall l : La)[\text{len } l = \text{sum } (\text{map } (\text{const } 1)\ l)]$

$\lambda x.\ 1$

$(\text{sum} \circ \text{map } (\text{const } 1))\ l$

$\text{len} = \text{sum} \circ \text{map } (\text{const } 1)$

const : α → α → α

const k x = k

-ou-

const k = λx. k

Θ? $(\forall f : \alpha \to \beta)\,[\,\text{len} \circ \text{map}\ f = \text{len}\,]$

DEMONS.

Seja $f : \alpha \to \beta$.

Seja $\ell : L\alpha$.

Basta $\text{len}\ (\text{map}\ f\ \ell) = \text{len}\ \ell$. [por quê?]

Sep casos ℓ:

CASO $[\,]$.

Calc: $\text{len}\ (\text{map}\ f\ [\,]) = [\text{map}.1\,]\ \text{len}\ [\,]$

DADOS

$f : \alpha \to \beta$

$\ell : L\alpha$

ALVO

$\text{len} \circ \text{map}\ f = \text{len}$

$\Updownarrow$

$(\forall \ell : L\alpha)\,[\,(\text{len} \circ \text{map}\ f)\ \ell = \text{len}\ \ell\,]$

CASO (x :: xs): len (map f (x :: xs)) ? = len (x :: xs)

CALC:

len (map f (x :: xs))

= [map.2 ...]

len (f x :: map f xs)

= [len.2]

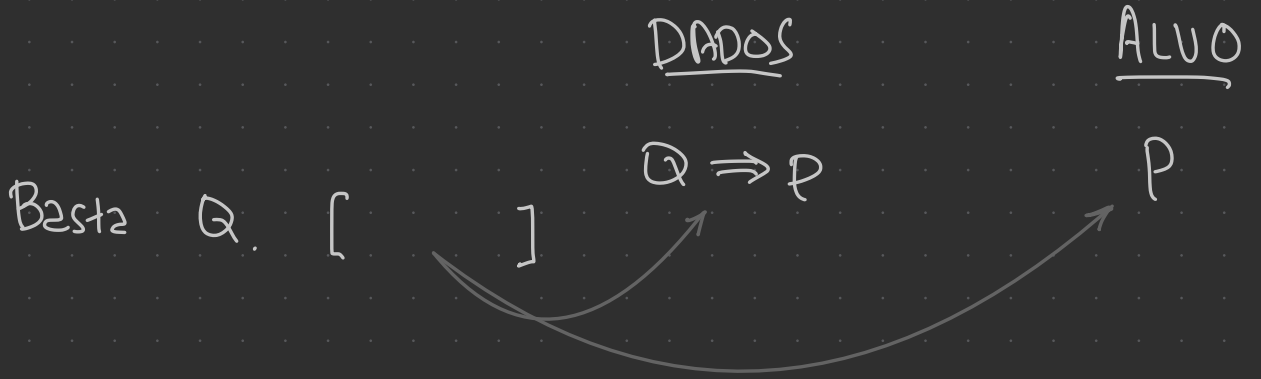
S (len (map f xs))

len (x :: xs)

= [len.2 ...]

S (len xs)

Basta len (map f xs) = len xs . [por quê?]


DADOS
ALUO
Q ⇒ P
P
Basta Q. [ ]

$$map : (\alpha \to \beta) \to L\alpha \to L\beta$$

```
data List α
  Nil  : List α
  Cons : α → List α → List α
```

```
map f []      = []
map f (x::xs) = f x :: map f xs
```

$f : \alpha \to \beta$
$x : \alpha$
$xs : L\alpha$
$map\ f\ xs : L\beta$

$\varphi : (l : L\alpha) \to len\ (map\ f\ l) = len\ l$

$\varphi\ [] = (demons.\ de\ len\ (map\ f\ []) = len\ [])$

$\varphi\ (x::xs) = (demons\ de\ len\ (map\ f\ (x::xs) = len\ (x::xs))$

↖ aqui temos acesso à $\varphi(xs)$ (H.I)

Unit & Empty

lec 13

2024-11-08

```
int  rand(void)
void print(str s)
void hello(void)
```

∴ `rand(_, _, _)`

chamar a função com

$$\frac{\text{rand} : \text{void} \to \text{int} \qquad \star : \text{Void}}{\text{rand}() : \text{int}}$$

1

```
data Unit
  * : Unit
```

```
data ()
  () : ()
```

? : Unit → β

```
f * = b
```

? : α → Unit

```
f x = *
```

$$1 \to \beta \cong \beta$$

0

```
data Empty
```

$? : Empty \to \beta$

$f : 0 \to \beta$

$? : \alpha \to Empty$

$f : 0 \to 0$

$0 \to \beta \cong ?$

Maybe

(Maybe : Type → Type)

```
data Maybe α
  Nothing : Maybe α
  Just    : α → Maybe α
```

Pra que serve isso?

```
data M α
  N : M α
  J : α → M α
```

..., Just 4, Just 0, Nothing : Maybe Nat

head : L α → α

```
safeHead : L α → M α
safeHead []      = Nothing
safeHead (x :: _) = Just x
```

Booleanismo

Não faça isso!

```
null : L α → Bool
null []  = True
null _   = False
```

```
if null l
   then ~~~
   else ~~~
```

aqui o programador toma cuidado para não usar head l, tail l, etc

aqui o programador usa head, tail para acessar tais "informações" da l

Functors

Lec 14
2024-11-13

$$\text{mapMaybe} : (\alpha \to \beta) \to M\,\alpha \to M\,\beta$$
$$\text{mapMaybe}\ f\ \text{Nothing}_\alpha = \text{Nothing}_\beta$$
$$\text{mapMaybe}\ f\ (\text{Just}_\alpha\ x) = \text{Just}_\beta\ (f\ x)$$

$$\text{map}\ \text{id} = \text{id}$$
$$\text{map}\ (f \circ g) = \text{map}\ f \circ \text{map}\ g$$

typeclass Functor (f : Type → Type)

fmap : (α → β) → (f α → f β)

+ as leis de Functor
- fmap id = id
- fmap (f ∘ g) = fmap f ∘ fmap g

instance Functor List
fmap = mapList

instance Functor Maybe
fmap = mapMaybe

---

definição inválida pois quebra as leis de functor

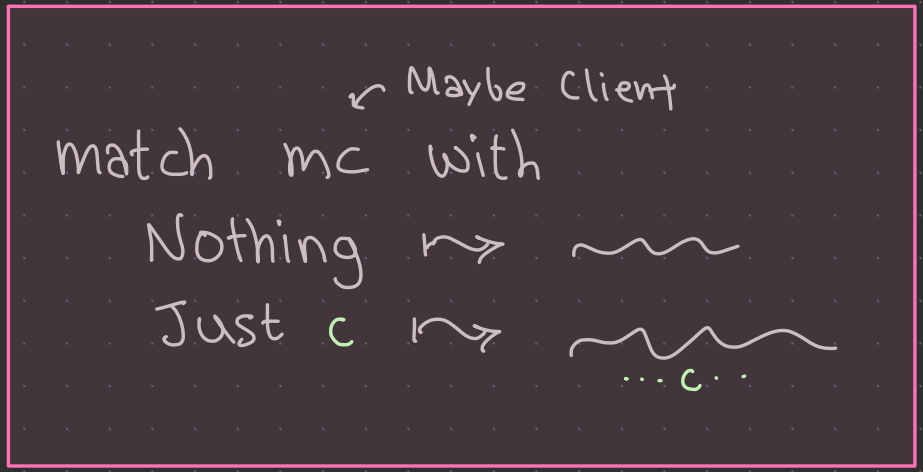
```
fmap f []       = []
fmap f [x]      = []
fmap f (x::x'::xs) = f x' :: fmap f (x'::xs)
```

fmap (·10) [1,2,3] = [10, 20]

fmap id [1,2] ≟ [1, 2]

fmap ((+1)∘(·2)) [10,20] ≟ (fmap (+1) ∘ fmap (·2)) [10,20]

Booleanismo



Either

```
data Either α β : Type
  Left  : α → Either α β
  Right : β → Either α β
```

```
Either : Ty → Ty → Ty
Either ε : Ty → Ty
```

```
instance Functor (Either ε)
  fmap : (α → β) → (Either ε α → Either ε β)
  fmap f (Left e)  = Left e
  fmap f (Right x) = Right (f x)
```

```
Either String Int
  Left "oi", Right 42
Either Error Int
  Left 42, Right 42, Left (WE 404)
```

(Error: Int; 42: CNF)

```
data Error
  ClientNotFound : Error
  DBOffline      : Error
  WebError       : Int → Error
```

$(M \circ L)\ \alpha$

$M\ (L\ \alpha)$

Nothing

Just []

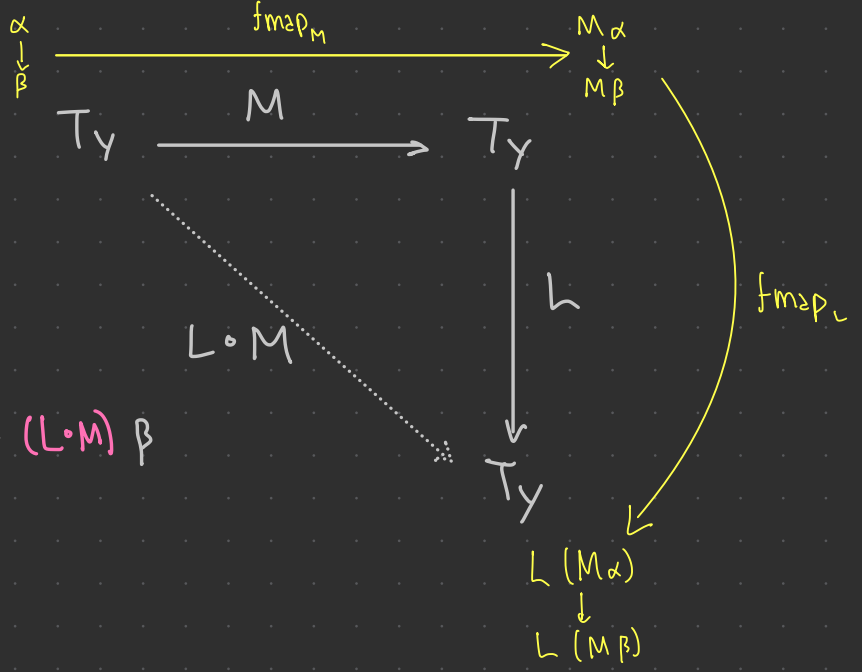
Just [1,2]

$(L \circ M)\ \alpha$

$L\ (M\ \alpha)$

[]

[J 1, N, J 8, N



$fmap : (\alpha \to \beta) \to (L \cdot M)\ \alpha \to (L \cdot M)\ \beta$

$fmap_{L \cdot M} = fmap_L \circ fmap_M$

Pairs

lec15
2024-11-22

$$\frac{\alpha : T_Y \qquad \beta : T_Y}{\alpha \times \beta \cdot T_Y}$$

$(\alpha, \beta) : T_Y$

$(,) : T_Y \to T_Y \to T_Y$

$$\frac{a : \alpha \qquad b : \beta}{(a,b) : \alpha \times \beta}$$

$(,) : \alpha \to \beta \to \alpha \times \beta$

```
data Pair α β
    MkPair : α → β → P α β
```

$\alpha \times \beta \equiv$ Pair α β

```
fst : Pair α β → α
fst (MkPair a b) = a

snd : Pair α β → β
snd (MkPair a b) = b
```

Records

$$c : \underset{\pi_1}{Str} \times \underset{\pi_2}{Str} \times \underset{\pi_3}{Nat} \times \underset{\pi_4}{Date}$$

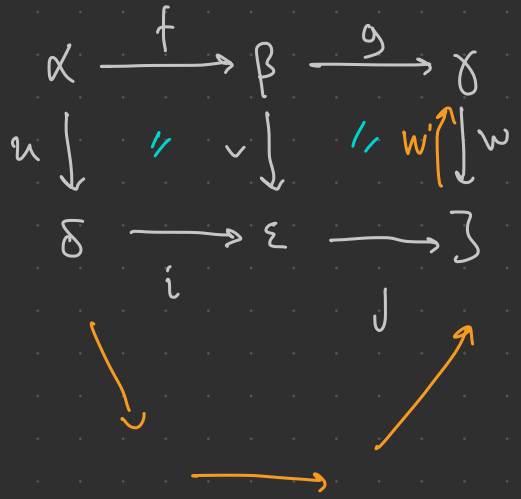
```
data Customer
  MkCustomer : Str → Str → Int → Date → Customer
```

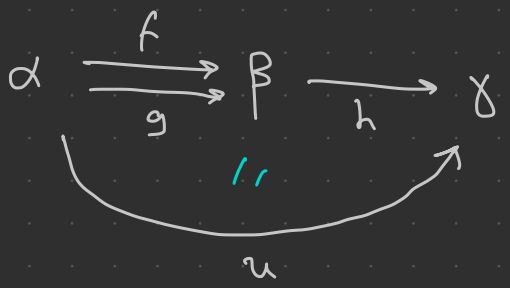
```
data Customer = Customer { name : Str
                         , email : Str
                         , id    : Int
                         , date  : Date
                         }
```

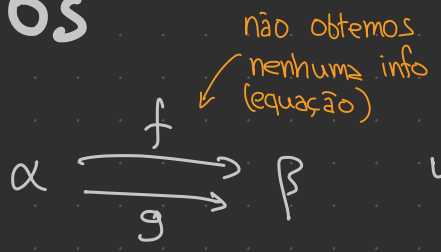
```
c = Customer { name = ".. "
             , email = "... " ...
             }
```

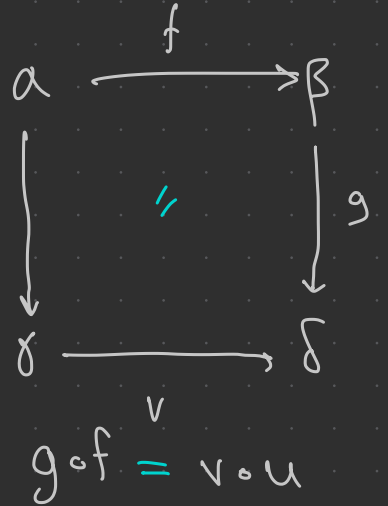
```
c name ≡ (.name) c
```

Diagramas comutativos





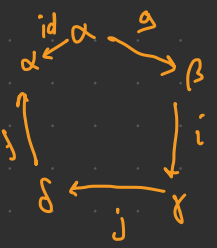


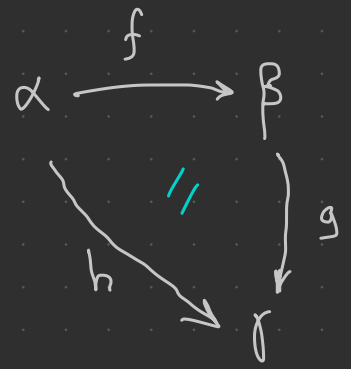


$g \circ f = v \circ u$

$w \circ g \circ f \overset{?}{=} j \circ i \circ u \overset{?}{=} j \circ v \circ f$

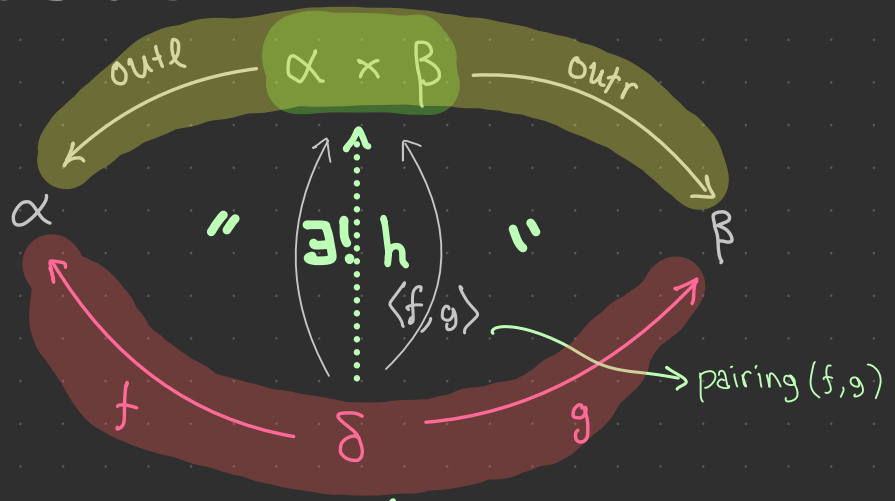
$= (w \circ g) \circ f = (j \circ v) \circ f = j \circ (v \circ f) = j \circ (i \circ u)$





$g \circ f = h$

Products



$$h\ x = (f\ x\ ,\ g\ x)$$

$f = outl \circ h$

$f\ x \overset{?}{=} (outl \circ h)\ x$

$= outl\ (h\ x)$

$= outl\ (f\ x\ ,\ g\ x)$

$g = outr \circ h$

(similar)

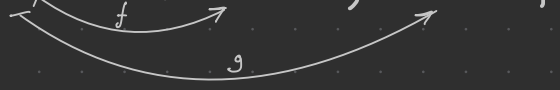
Como entramos no $\alpha \times \beta$?

$h : \delta \longrightarrow \alpha \times \beta$

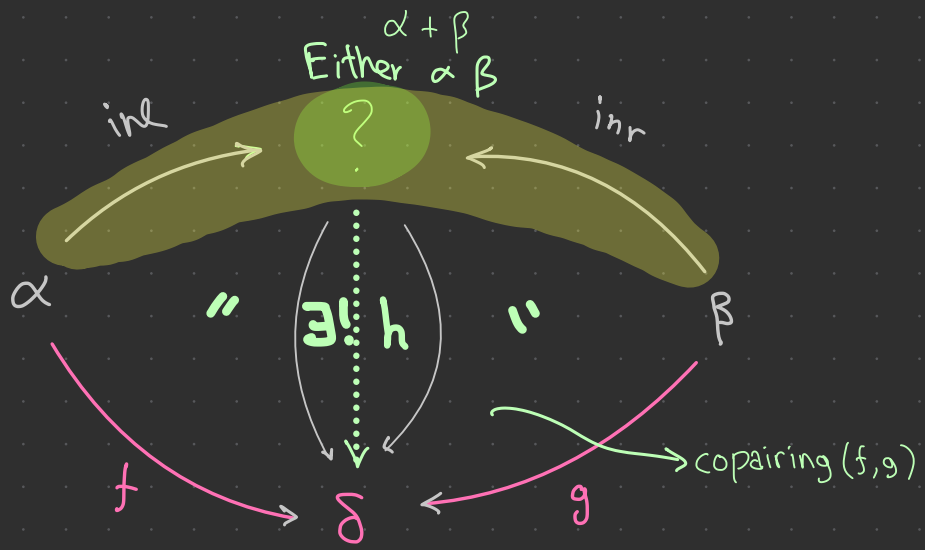
$outl \circ h = f$

$outr \circ h = g$

$h\ x = (\ldots x \ldots\ ,\ \ldots x \ldots)$



Coproducts (sums)



Coentramos no

Como saimos do $\alpha + \beta$?

$$h : \alpha + \beta \longrightarrow \delta$$

$$h \circ inl = f \qquad h\ (inl\ x) = f\ x$$

$$h \circ inr = g \qquad h\ (inr\ y) = g\ y$$

Indução em listas

lec16

2024-11-27

```
(++) : L α → L α → L α
[]     ++ ys  = ys
(x :: xs) ++ ys  = x :: (xs ++ ys)
```

[2,3,7,8]

```
xs ++ []     = xs
xs ++ (y :: ys) = insertAt
                    (length xs)
                    y
                    (xs ++ ys)
```

[1,2,3,7,8]

[1,2,3] [7,8]

xs ++ ys

[1,2,3,8]

```
[1,2,3] ++ [7,8]
  = 1 :: ([2,3] ++ [7,8])
  = 1 :: 2 :: ([3] ++ [7,8])
  = 1 :: 2 :: 3 :: ([] ++ [7,8])
  = 1 :: 2 :: 3 :: [7,8]
  ≡ [1,2,3,7,8]
```

8. $\underbrace{\text{sum}\ (xs \mathbin{+\!\!+} ys) = \text{sum}\ xs + \text{sum}\ ys}_{\varphi(xs)}$

Sejam xs, ys : L Nat.

Por indução no xs

CASO []:

$\varphi([]) \stackrel{?}{=}$

ALVO: sum ([] ++ ys) = sum [] + sum ys

calc: sum ([] ++ ys)
= sum ys
sum [] + sum ys
= 0 + sum ys
...
= sum ys

match xs with

[] ↝ ...

(k::ks) ↝ ...

CASO $(k :: ks)$: $\varphi(k :: ks)$

ALVO: $sum\ ((k::ks) ++ ys) \stackrel{?}{=} sum\ (k::ks) + sum\ ys$

calc:

$sum\ ((k::ks) ++ ys)$

$= sum\ (k :: (ks ++ ys))$

$= k + sum\ (ks ++ ys)$

$= k + (sum\ ks + sum\ ys)$ [H.I.]

$= (k + sum\ ks) + sum\ ys$

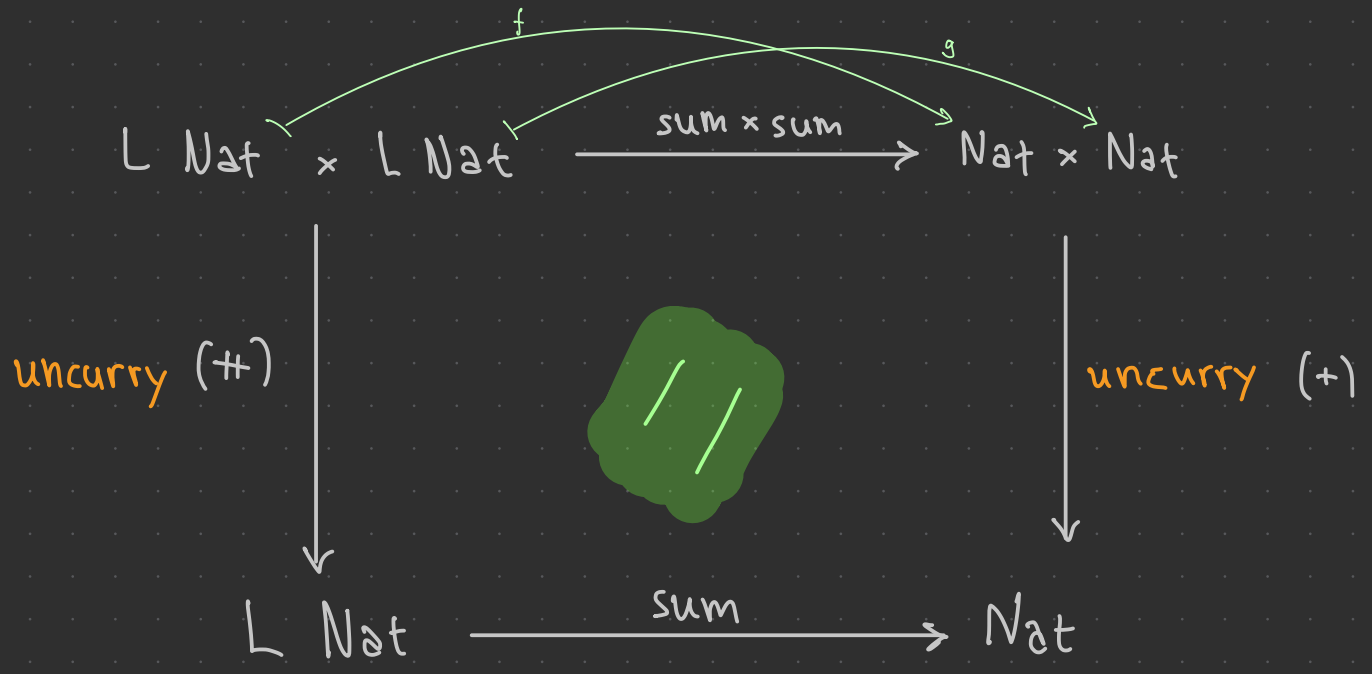
$= sum\ (k :: ks) + sum\ ys$

Presente da indução: $\varphi(ks)$

$sum\ (ks ++ ys) = sum\ ks + sum\ ys$

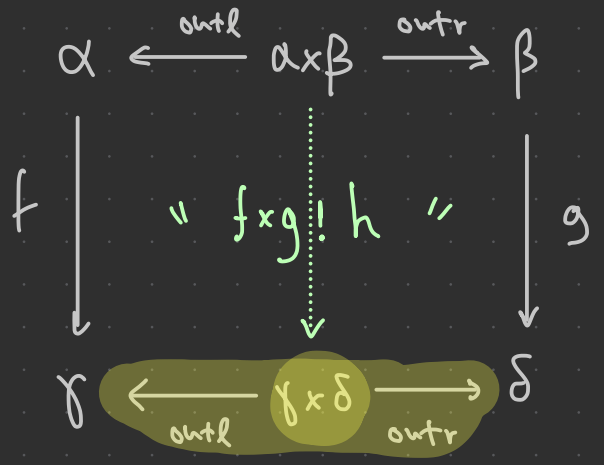
Diagramas comutativos

$$\text{sum}\ (xs +\!\!+\ ys) = \text{sum}\ xs + \text{sum}\ ys$$



$$\text{Sum} \circ \text{uncurry}\ (+\!\!+) = \text{uncurry}\ (+) \circ (\text{sum} \times \text{sum})$$

Funções de graça



é um produto (portanto fornece uma maneira de chegar nele)

$$h\ (a,b) \stackrel{def}{=} (f\ a\ ,\ g\ b)$$

$$f \times g \stackrel{def}{=} pairing\ (f \circ outl,\ g \circ outr)$$

$$\equiv \langle f \circ outl\ ,\ g \circ outr \rangle$$

$$pairing : (\delta \to \alpha) \times (\delta \to \beta) \to (\delta \to \alpha \times \beta)$$

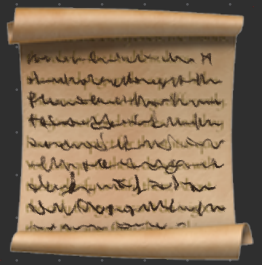
$$\frac{\delta \to \alpha \qquad \delta \to \beta}{\delta \to \alpha \times \beta}\ pairing$$

$$\frac{\alpha \to \gamma \qquad \beta \to \delta}{\alpha \times \beta \to \gamma \times \delta}\ cross$$

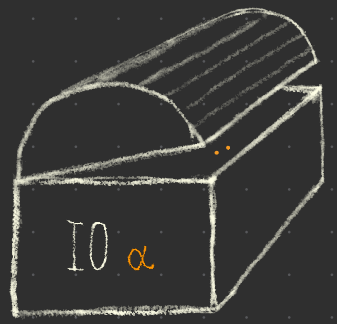
OI, IO

lec17

2024-11-29



: IO α



pure : α → IO α

```
do
  a ← ioa : IO α
  iob     : IO β
  c ← ioc : IO γ
  ⋮         ⋮
  ioz     : IO ω
```

: IO ω

ghci :

REPL

ou

REiIO1tXLeiIOαXLePL ?

Functor IO

lec18

2024-12-04

```
Functor (IO : Ty → Ty)
  fmap : (α → β) → (IO α → IO β)
  fmap f ax =
    do x ← ax
       let y = f x
       return y
       pure
```

```
fmap f ax =
  do x ← ax
     return x
     pure
     return $ f x
     x' ← ax
     return $ f x'
     pure
```

? • `fmap id = id`

• `fmap (f . g) = fmap f . fmap g`

`(.)` precedência alta, assoc-L

`($) : (α → β) → α → β` precedência baixa, assoc-R

FAM · Functor, Applicative, Monad

typeclass Functor (f : Ty→Ty)

fmap : (α → β) → (f α → f β)

(+ Leis)

typeclass (Functor f) ⇒ Applicative (f : Ty→Ty)

pure : α → f α

(⊛) : f (α → β) → (f α → f β)

splat

<*>

(+ Leis (?))

typeclass (Applicative m) ⇒ Monad (m : Ty → Ty)

(?)

Instances de Applicative

Applicative IO

```
pure = pure_IO
af ⊛ ax = do f ← af
             x ← ax
             pure (f x)
```

```
do f ← af
   fmap f ax
```

Applicative Maybe

```
pure x = Just x
Nothing ⊛ _ = Nothing
(Just f) ⊛ Nothing = Nothing
(Just f) ⊛ (Just x) = Just (f x)
```

```
pure = Just
Nothing ⊛ _ = Nothing
Just f ⊛ mx = fmap f mx
```

Applicative list

```
pure x = [x]
fs ⊛ xs =
[] ⊛ _ = []
_ ⊛ [] = []
(f :: fs) ⊛ (x :: xs) = f x :: (fs ⊛ xs)
```

?

Applicative list

...?

Interpretações computacionais

lec19
2024-12-06

valor ambíguo (indeterminado)

Applicative List

```
pure x = [x]
fs ⊛ xs = [f x | f ← fs, x ← xs]
(⊛) = ...point-free...
```

valor ao longo do tempo (temporal)

Applicative List

```
pure x = [x, x, ...]
(⊛) = pw ($)
```


```
[(+1), (·3)] ⊛ [42, 0, 3, 3]
  = [43, 1, 4, 4, .., 9]
```

```
[(+1), (·3), (+2), (+1)]
⊛
[42, 0, 3, 3, 42]
=
[43, 0, 5, 4]
```

Cartesian product

```
cp : L α × L β → L (α × β)
cp [(+1), (·3)] [42, 0, 3, 3] = [((+1), 42), ((+1), 0), ..., ((·3), 3)]
                              : List ((α → β) × α)
cp = ...
```

SMG : Semigroup, Monoid, Group

typeclass Semigroup (s : Ty)

(◊) : s → s → s

+ Leis : (◊)-assoc.

typeclass (Semigroup m) ⇒ Monoid (m : Ty)

mempty : m

+ Leis : mempty-idR , mempty-idL

typeclass (Monoid g) ⇒ Group (g : Ty)

inverse : g → g

+ Leis : (inverse x) ◊ x = mempty

x ◊ (inverse x) = mempty

newtype: cópias isomórficas de tipos

newtype

~~data~~ Sum = Sum Nat

~~data~~ Product = Product Nat

newtype

ou, usando Records:

toNat, fromSum, ...

newtype Sum = Sum { getSum : Nat }

newtype Product = Product { getProduct · Nat }

```
instance Semigroup ~~Nat~~ Sum
  (◊) = (+)

instance Semigroup ~~Nat~~ Product
  (◊) = (·)
```

Applicative laws

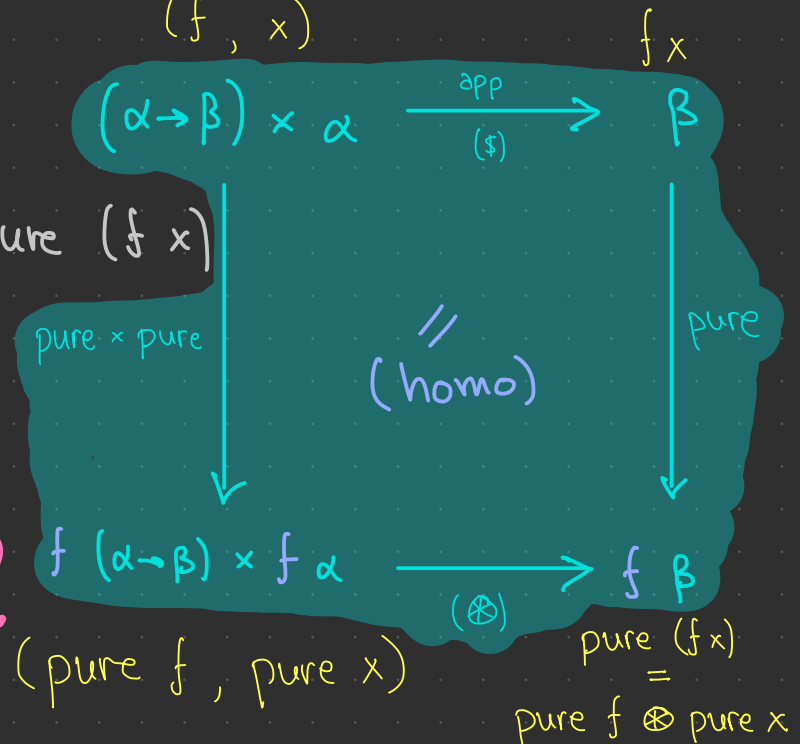
lec20
2024-12-11

id: $\underbrace{\text{pure}\ \text{id}}_{f\ (\alpha\to\alpha)} \circledast v = v$ ($\text{id} : \alpha\to\alpha$; $v : f\ \alpha$; $f\ \alpha$)

homo: $\text{pure}\ f \circledast \text{pure}\ x = \text{pure}\ (f\ x)$

?: $u \circledast \text{pure}\ x = ?$

?: $u \circledast (v \circledast w) = ?$



Demonstração: com vs sem pontos

0. $\langle f, g\rangle = \langle h, k\rangle \Rightarrow f = h \;\&\; g = k$

ALVO : $f = h \quad (\Leftrightarrow (\forall x : \delta)[\, f\,x = h\,x\,])$

Seja $x : \delta$. ALVO : $f\,x = g\,x$

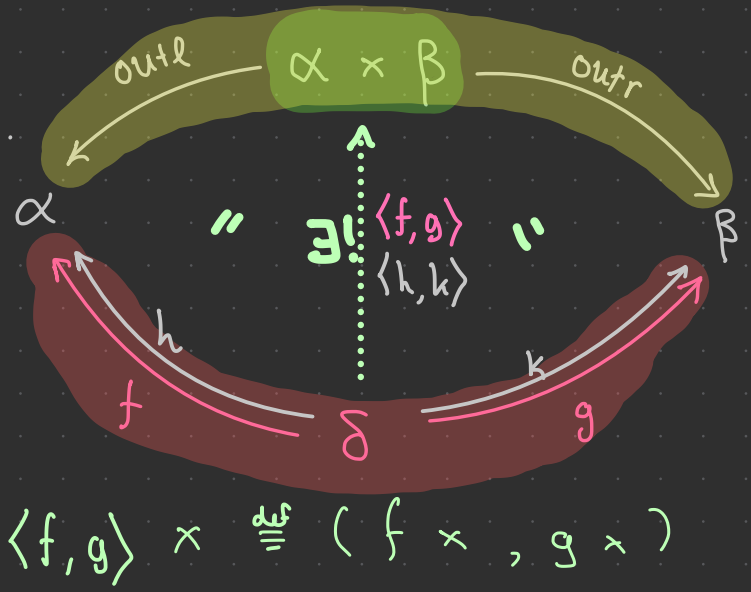
Temos $\langle f, g\rangle\, x = \langle h, k\rangle\, x$.

Logo $(f\,x, g\,x) = (h\,x, k\,x)$.

Logo $f\,x = h\,x$.

ALVO : $g = k$

Similar.



do laws

```
do   x ← p
     return x
```

$\rightsquigarrow$ (do) p

```
do   x ← return e
     f x
```

: 10 β

$\rightsquigarrow$ (do) f e

f : α → 10 β

vini : 10 α × (α → 10 β) × (β → 10 δ) → 10 γ

vini p f g =

```
do   y ← do   x ← p
          f x
     g y
```

$\rightsquigarrow$

```
do   x ← p
     y ← f x
     g y
```

: 10 γ

Eficiência "por indução"

lec21

2024-12-13

```
rev : L α → L α
rev []      = []
rev (x::xs) = rev xs ++ [x]

(++) : L α → L α → L α
[] ++ ys      = ys
(x::xs) ++ ys = x :: (xs ++ ys)
```

($\overbrace{xs}^{n} ++ \overbrace{ys}^{m}$ precisa $n+1$ passos)

```
rev [1,2,3,4]
= rev [2,3,4] ++ [1]
= (rev [3,4] ++ [2]) ++ [1]
= ((rev [4] ++ [3]) ++ [2]) ++ [1]
= (((rev [] ++ [4]) ++ [3]) ++ [2]) ++ [1]
= (([] ++ [4]) ++ [3]) ++ [2]) ++ [1]
≡ [4,3,2,1]
```

$n+1$

$1 + \cdots + n = \frac{n(n+1)}{2}$

$$n+1 + \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{3}{2}n + 1 = O(n^2)$$

WISH: $revcat : L\,\alpha \to L\,\alpha \to L\,\alpha$

$(\forall xs, ys)\,[\,revcat\ xs\ ys \overset{\text{não def}}{=} rev\ xs +\!\!+ ys\,]$

THEN: $rev\ xs \overset{def}{=} revcat\ xs\ []$

$revcat\ []\ ys = ?$

$revcat\ (x::xs)\ ys = ?$

"Por indução."

CASO `[]`:

$$
\begin{aligned}
&revcat\ []\ ys \\
&\quad = rev\ []\ +\!\!+\ ys \\
&\quad = []\ +\!\!+\ ys \\
&\quad = ys
\end{aligned}
$$

CASO `(x::xs)`:

$$
\begin{aligned}
&revcat\ (x::xs)\ ys \\
&\quad = rev\ (x::xs)\ +\!\!+\ ys \\
&\quad = ((rev\ xs) +\!\!+ [x]) +\!\!+ ys \\
&\quad = (rev\ xs) +\!\!+ ([x] +\!\!+ ys) \\
&\quad \overset{\cdot\cdot}{=} revcat\ xs\ ([x] +\!\!+ ys) \\
&\quad \overset{\cdot\cdot}{=} revcat\ xs\ (x : ys)
\end{aligned}
$$

```
revcat : L α → L α → L α
revcat []     ys = ys
revcat (x::xs) ys = revcat xs (x : ys)

rev xs =def= revcat xs []
```

```
rev [1,2,3,4]
  = revcat [1,2,3,4] []
  = revcat [2,3,4] (1]
  = revcat [3,4] [2,1)
  = revcat (4] [3,2,1)
  = revcat [] [4,3,2,1)
  = [4,3,2,1)
```

$n+1$ $O(n)$ n ← Linear

n^2, $n^{3/2}$, $n^{1+\varepsilon}$ } polinomial

Sorting (Ord α) ⇒ ...

```
sort   : (Ord α) ⇒ L α → L α

sorted : (Ord α) ⇒ L α → Bool
sorted []       = True
sorted [_]      = True
sorted (x :: x' :: _) = x ≤ x' & sorted xs
                        ~~~~~~~~~~~~~~~~~
                        x ≤ head xs & sorted xs
                  (xs = x' :: _)
```

```
data Ordering = LT | EQ | GT

typeclass Ord α
    compare : α → α → Ordering
    (<), (≤), ...
```

Corretude da sort: $(\forall l)[\ \text{sorted}\ (\text{sort}\ l) = \text{True}\ ]$

Merge sort (vN)

```
msort [] = []
msort [x] = [x]
```

?

```
msort xs = merge (msort us) (msort vs)
  where (us, vs) = halve xs
```

```
where
  us = take l xs
  vs = drop l xs
  l  = length xs / 2
```

```
halve : L α → (L α × L α)
halve []  = ([], [])
halve [x] = ([x], [])
halve (x :: x' :: xs) = ?
```

```
merge : L α → L α → L α
merge [] vs = vs
merge us [] = us
merge (u :: us) (v :: vs)
  | u ≤ v       = u :: merge us (v :: vs)
  | otherwise   = v :: merge (u :: us) vs
```

HW

Quick sort

7 2 0 1 3 2 8 9 7

(<7): 2 0 1 3 2

(≥7): 8 9 7

```
qsort []     = []
qsort [x]    = [x]        ?
qsort (p :: xs) = qsort small ++ [p] ++ qsort large
```

p ← pivot

where

```
small = filter (< p) xs
large = filter (≥ p) xs
```

where

```
(small, large) = ?
```

Insertion sort (HW)

```
isort []     = []
isort (x :: xs) = ...insert ..
```

insert : α → L α → L α

Fold List α

lec22
2024-12-18

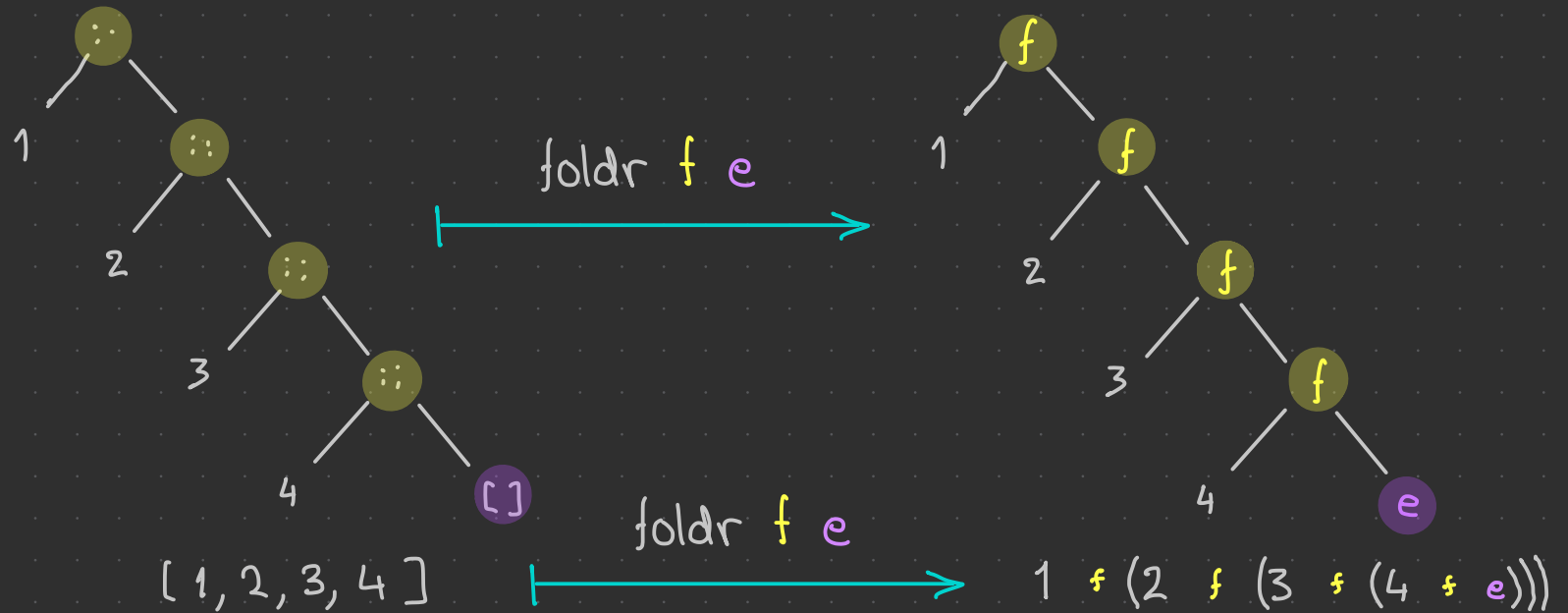
Cons : α → Lα → Lα

foldr : (α → β → β) → β → (Lα → β)

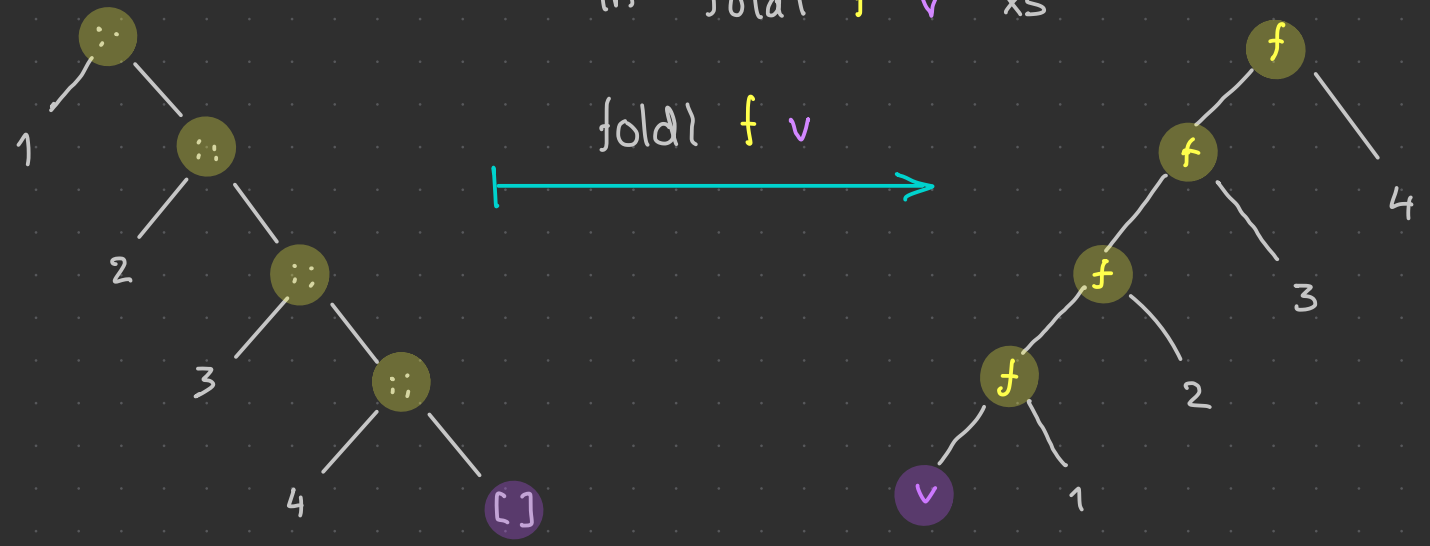
Nil : Lα

foldr f e [] = e

foldr f e (x::xs) = x f (foldr f e xs)



```
foldl : (β → α → β) → β → (L α → β)
foldl f v []      = v
foldl f v (x::xs) = let v' = v f x
                    in foldl f v' xs
```



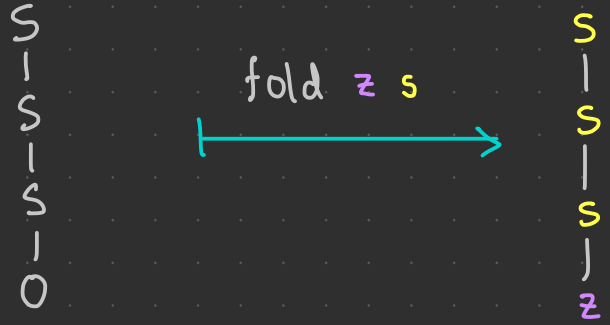

```
foldl f v [1, 2, 3, 4]  ⟼  (((v f 1) f 2) f 3) f 4
```

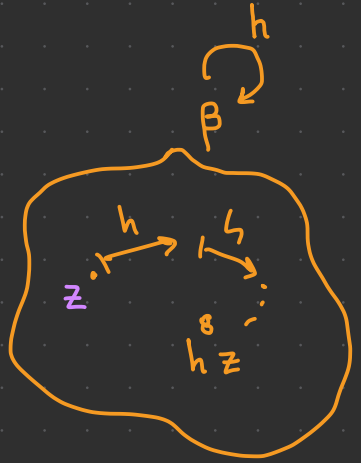
fold Nat

$$\underset{\substack{\text{O} : \text{Nat} \\ \text{S} : \text{Nat} \to \text{Nat}}}{\text{fold} : \beta \to (\beta \to \beta) \to (\text{Nat} \to \beta)}$$

$$\text{fold}\ z\ s\ \text{O} = z$$

$$\text{fold}\ z\ s\ (\text{S}\ n) = s\ (\text{fold}\ z\ s\ n)$$





recursão ajuda sair dum tipo indutivo

$f : \mathbb{N}at \to \beta$

$f\ 0 = z$

$f\ (S\ n) = \ldots (f\ n) \ldots$

$h\ (f\ n)$

recursão não ajuda entrar num tipo indutivo

$g : \alpha \to \mathbb{N}at$

(teaser: corecursão ajuda entrar num tipo coindutivo)

$x \xmapsto{h} \ldots x \ldots$

$fact\ 0 = 1$

$fact\ (S\ n) = S\ n \cdot fact\ n$

$= \ldots\ fact\ n \ldots$

$= h\ (fact\ n)$

where $h\ x = S\ n \cdot x$

$fact = ?$

$fact = fold\ 1\ (\lambda x.\ \underbrace{S\ n}_{?} \cdot x)$

Listas como Applicatives

```
newtype Ambiguous α =
  Ambiguous (L α)
```

```
newtype Temporal α =
  Temporal (L α)
```

```
instance Applicative T
  pure x = T [x, x, ...]        -- always x / forever x
  T fs ⊛ T xs = T (pw ($) fs xs)
```

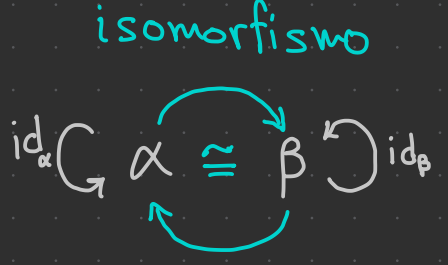
```
instance Applicative A
  pure x = A [x]                -- surely x
  A fs ⊛ A xs = A [f x | f ← fs, x ← xs]
```

```
pure f ⊛ ax
  = [f, f, f, ...]
    ⊛
    [x1, x2, ...?; xn]
  = [f x1, f x2, ...?; f xn]
```

```
fmap f ax = f <$> ax
```

```
pure f ⊛ ax
  = [f]
    ⊛
    [x1, x2, ...?; xn]
  = [f x1, f x2, ...?; f xn]
```

Cópias isomórficas de tipos




```
type Tagged α = String × α
```

```
f : Tagged α → Tagged β  ≡  f : String × α → String × β
```

```
type Balance = Integer
type Days    = Integer
```

```
type Forest α = List (Tree α)
```

```
f : Balance → Days → Balance
```

```
data Balance = MkBalance Integer
```

⊥, MB ⊥, MB 0, MB 1, MB 2, ...

HW

```
newtype Balance = MkBalance Integer
```

MB ⊥, MB 0, MB 1, MB 2, ...

Eficiência por indução

lec23
2025-01-10

```
flat : Tree α → List α
flat (T x)   = [x]
flat (F l r) = flat l ++ flat r
```

Gostaríamos de ter uma flat′ t.q.

```
flat′ t xs  =  flat t ++ xs
```

pois então ...

```
flat t ≝ flat′ t []

flat′ : Tree α → List α → List α
flat′ (T x) xs   =  ?   (BASE)
flat′ (F l r) xs =  ?   (PASSO INDUTIVO)
```

BASE:

```
flat' (T x) xs
  = flat (T x) ++ xs
  = [x] ++ xs
  = x :: xs
```

PASSO INDUTIVO:

```
flat' (F l r) xs
  = flat (F l r) ++ xs
  = (flat l ++ flat r) ++ xs
  = flat l ++ (flat r ++ xs)
  = flat l ++ (flat' r xs)
  = flat' l (flat' r xs)
```

Finalmente...

```
flat t ≝ flat' t []
  where
    flat' : Tree α → List α → List α
    flat' (T x) xs = x :: xs
    flat' (F l r) xs = flat' l (flat' r xs)
```

```
eval : Expr → M Int
eval (Val n)   = J n
eval (Div u v) =
    case eval u of
      N ↝ N
      J x ↝ case eval v f
              N ↝ N
              J y ↝ safediv x y
```

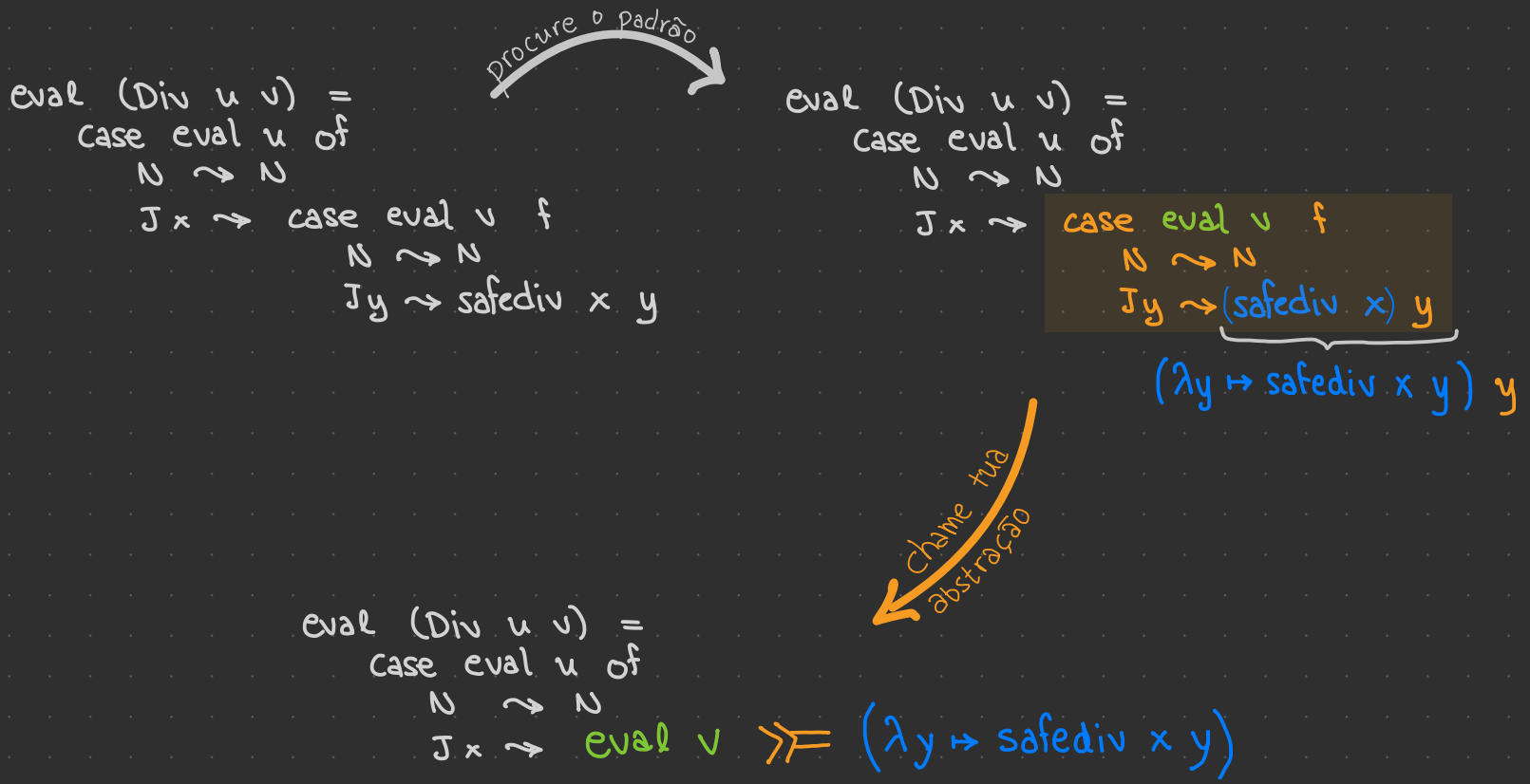
perceba o padrão

```
case [eval ...] of
  N ↝ N
  J x ↝ [faça algo com] x
```

DRY!

abstraia (parametrize)

```
(>>=) : M α → (α → M β) → M β
mx >>= f = case [mx] of
             N ↝ N
             J x ↝ [f] x
```


eval (Div u v) =
case eval u of
N ⇝ N
J x ⇝ case eval v of
N ⇝ N
J y ⇝ safediv x y
procure o padrão
eval (Div u v) =
case eval u of
N ⇝ N
J x ⇝ case eval v of
N ⇝ N
J y ⇝ (safediv x) y
(λy ↦ safediv x y) y
chame tua abstração
eval (Div u v) =
case eval u of
N ⇝ N
J x ⇝ eval v >>= (λy ↦ safediv x y)

```
eval (Div u v) =
  case eval u of
    N x ⇝ N
    J x ⇝ eval v >>= (λy ↦ safediv x y)
```

```
eval (Div u v) =
  case eval u of
    N x ⇝ N
    J x ⇝ eval v >>= (λy ↦ safediv x y)
```

$$(\lambda x \mapsto \text{eval } v \gg\!= (\lambda y \mapsto \text{safediv } x\ y))\ x$$

```
eval (Div u v) =
  eval u >>= (λx ↦ eval v >>= (λy ↦ safediv x y))
```

Notação do

```
eval u >>= (λx ↦ eval v >>= (λy ↦ safediv x y))
```

```
eval u >>= (λx ↦
eval v >>= (λy ↦
safediv x y))
```

```
do  x ← eval u
    y ← eval v
    safediv x y
```

Monads


```
fmap   : (α → β) → (m α → m β)
(⊛)    : m (α → β) → m α → m β
pure   : α → m α

return : α → m α
(>>=)  : m α → (α → m β) → m β      -- bind
join   : m (m α) → m α
(>=>)  : (α → m β) → (β → m γ) → (α → m γ)
```

Kleisli composition

Monad

typeclass (Applicative m) ⇒ Monad (m : Ty → Ty)

um dos

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

$$\text{join} : m\ (m\ \alpha) \to m\ \alpha$$

$$(>=>) : (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

$(>=>)$ from join : $(f >=> g)\ a =$ ... HW ...

⋮

HW

list comprehension & do

Cartesian product

```
cp : L α → L β → L (α × β)
cp xs ys = [ (x,y) | x ← xs, y ← ys ]

cp xs ys = do x ← xs
              y ← ys
              pure (x, y)
```

lec24

2025-01-15

lift

$$\mathrm{fmap}_f : (\alpha \to \beta) \to (f\,\alpha \to f\,\beta)$$

$$\mathrm{ap}_m := (\circledast_a) \equiv \mathrm{splat}_a : a\,(\alpha \to \beta) \to (a\,\alpha \to a\,\beta)$$

bind

$$? \equiv (=\!\!\ll_m) : (\alpha \to m\,\beta) \to (m\,\alpha \to m\,\beta)$$

$$(\gg\!\!=_m) : m\,\alpha \to (\alpha \to m\,\beta) \to m\,\beta$$

$$\mathrm{join}_m : m\,(m\,\alpha) \to m\,\alpha$$

$$\mathrm{return}_m := \mathrm{pure}_a : \alpha \to a\,\alpha$$

kleisli

$$(\rightarrowtail_m) : (\alpha \to m\,\beta) \to (\beta \to m\,\gamma) \to (\alpha \to m\,\gamma)$$

$$(\leftarrowtail_m) : (\beta \to m\,\gamma) \to (\alpha \to m\,\beta) \to (\alpha \to m\,\gamma)$$

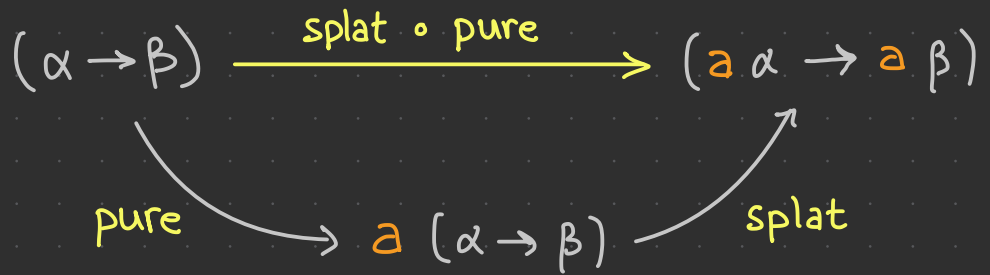
$$(\gg_m) := (*_a) : a\,\alpha \to a\,\beta \to a\,\beta$$

$$\mathrm{void}_f : f\,\alpha \to f\,1$$

$$? : m\,\alpha \to \alpha$$

Tip of the day: use Hoogle!

Applicative ⇒ Functor



E o resto?

HW

do vs. do

Applicative do:

```
do  x ← mx
    y ← my
    z ← mz
    pure $ f x y z
```

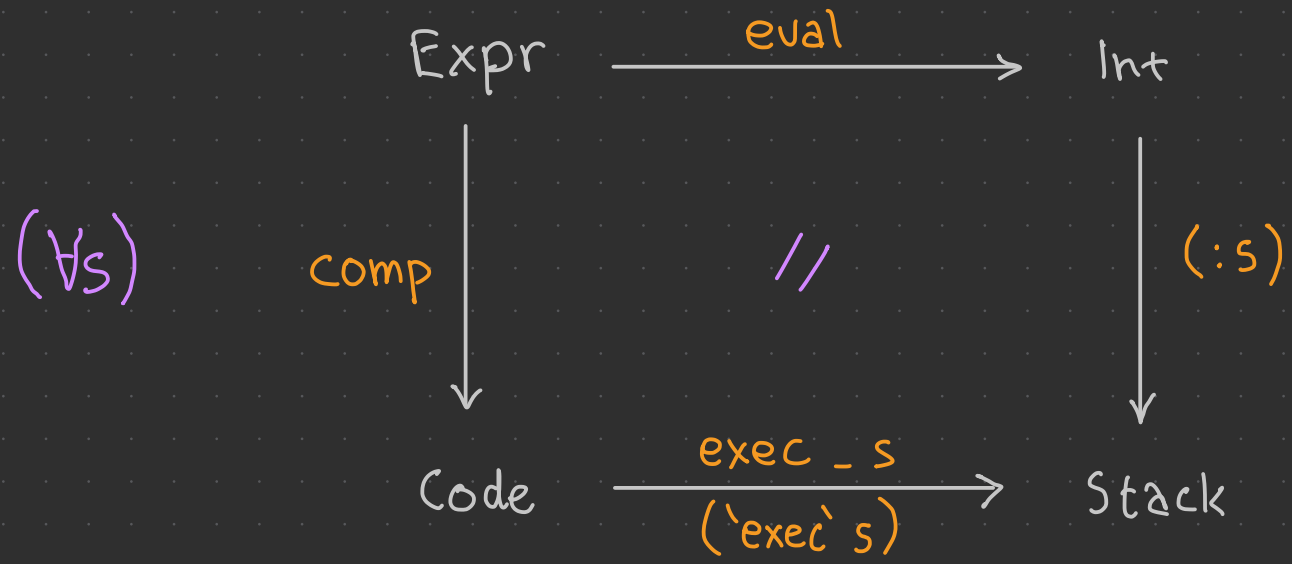
Monadic do:

```
do  x ← mx
    y ← u x
    z ← v x y
    f x y z
```

Compiler correctness

```
exec (comp e) [] = [eval e]
```

```
exec (comp e) s = eval e : s
```



Θ. exec (comp e) s = eval e : s

Indução no e.

CASO (Val n):

exec (comp (Val n)) $\stackrel{?}{=}$ eval (Val n) : s

= =

exec [PUSH n] s n : s

=

exec [] (n : s)

=

n : s

lec25

2025-01-17

```
exec : Code → Stack → Stack
exec [] s = s
exec (PUSH n : c) s = exec c (n : s)
exec (ADD : c) (n : m : s) = exec c ((m+n) : s)

comp : Expr → Code
comp (Val n) = [PUSH n]
comp (Add u v) = (comp u ++ comp v) ++ [ADD]

eval : Expr → Int
eval (Val n) = n
eval (Add u v) = eval u + eval v
```

Θ. exec (comp e) s = eval e : s

CASO (Add u v):

exec (comp (Add u v)) s $\stackrel{?}{=}$ eval (Add u v) : s

=

exec ((comp u ++ comp v) ++ [ADD]) s

=

exec (comp u ++ (comp v ++ [ADD])) s

= [Lemma]

exec (comp v ++ [ADD]) (exec (comp u) s)

= [H.I.]

exec (comp v ++ [ADD]) (eval u : s)

= [mesmo lemma]

exec [ADD] (exec (comp v) (eval u : s))

= [H.I.]

exec [ADD] (eval v : eval u : s)

=

exec [] ((eval u + eval v) : s)

=

(eval u + eval v) : s

=

eval (Add u v) : s

Lemma. exec (c ++ d) s = exec d (exec c s).

Indução no c.

CASO [].

⋮

CASO (PUSH n : c).

⋮

CASO (ADD : c).

exec ((ADD : c) ++ d) s

=

exec (ADD : (c ++ d)) s

= [Assumo s casa com m : n : s'] (TdC)

exec (ADD : (c ++ d)) (m : n : s')

=

exec (c ++ d) ((n + m) : s')

= [H.I.]

$$
\begin{aligned}
& exec\ d\ (exec\ c\ ((n+m) : s')) \\
&= \\
& exec\ d\ (exec\ (ADD : c)\ (m : n : s')) \\
&= \\
& exec\ d\ (exec\ (ADD : c)\ s)
\end{aligned}
$$

BRONCAS:

- comp é lento
- demonstrar sua corretude também
- ... e ainda mais acabou dependendo do [TdC]

Eficiência por indução

(i) redefinir comp

comp e $\overset{\text{def}}{=}$ ... comp' ...

onde comp' : Expr → Code → Code

comp' ...

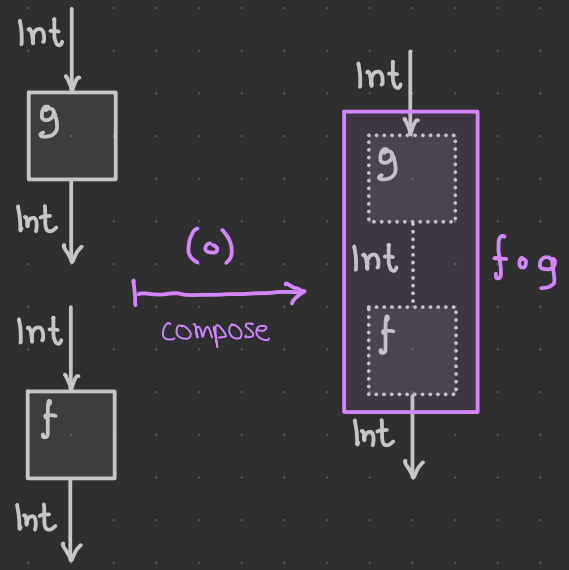
Dica: comp' e c $\overset{\ddot{\smile}}{=}$ comp e ++ c

(ii) demonstre

exec (comp e) s = eval e : s

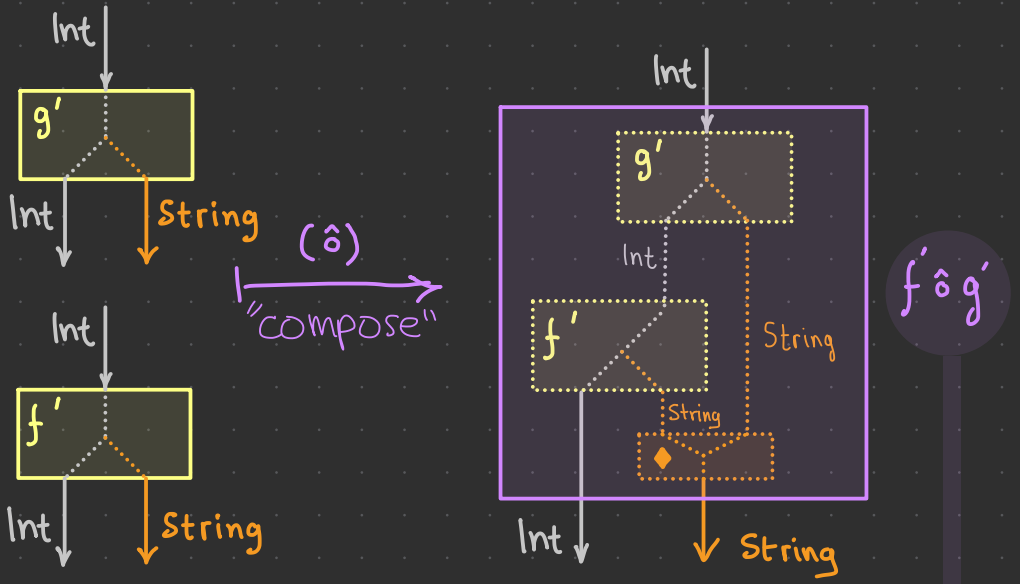
Functions

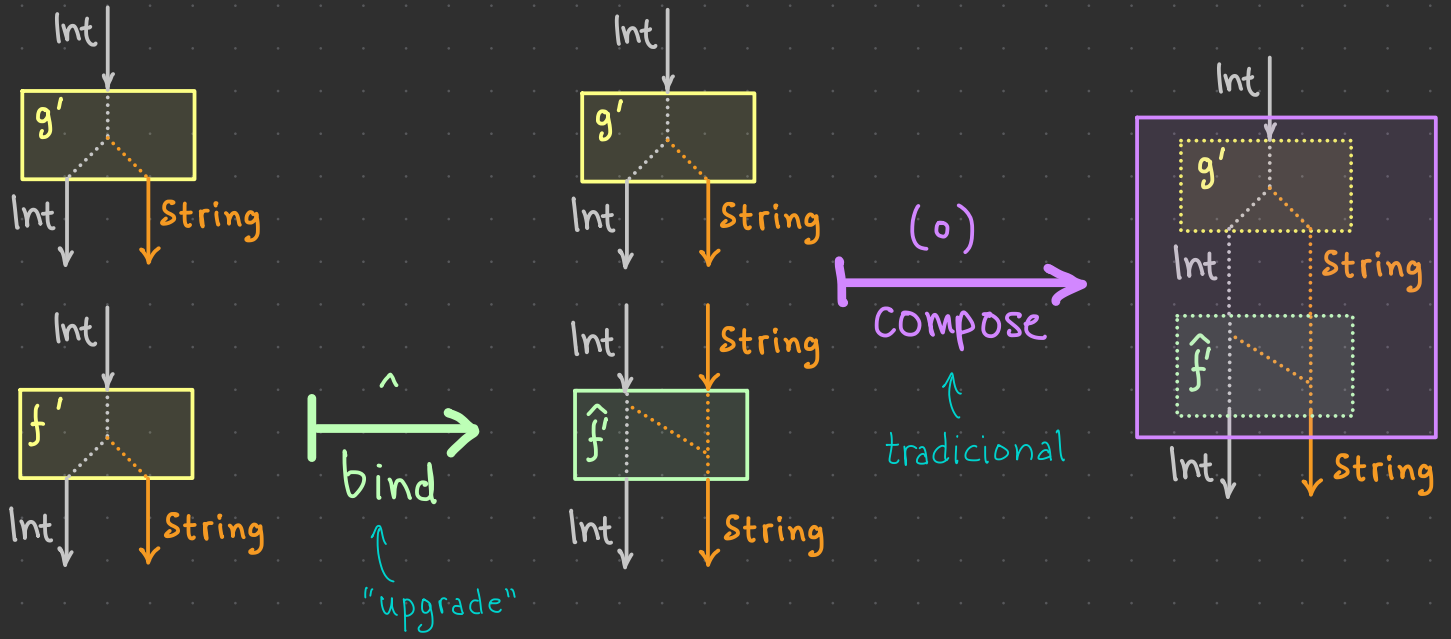
$f, g : Int \to Int$



Debuggable functions

$f', g' : Int \to Int \times String$





$$f' \hat{\circ} g' \overset{\text{def}}{\equiv} \hat{f}' \circ g'$$

Definições

$(\hat{\_} \equiv)$ bind : $(\text{Int} \xrightarrow{f'} \text{Int} \times \text{String}) \to (\text{Int} \times \text{String} \xrightarrow{\hat{f}'} \text{Int} \times \text{String})$

f

$$\underbrace{\text{bind}\ f'}_{\hat{f}'}\ (x, s) \overset{\text{def}}{\equiv} \text{let}\ (\overbrace{v_f}^{\text{value from } f}, \overbrace{m_f}^{\text{message from } f}) = f'\ x \ \text{in}\ (v_f, s +\!\!+ m_f)$$

Kleisli composition

$(\hat{\circ})$: $(\text{Int} \to \text{Int} \times \text{String}) \to (\text{Int} \to \text{Int} \times \text{String}) \to (\text{Int} \to \text{Int} \times \text{String})$

$f' \ \hat{\circ}\ g' \overset{\text{def}}{\equiv} \hat{f}' \circ g'$

$(f' \ \hat{\circ})\ g = (\hat{f}' \circ)\ g$

$(\hat{\circ})\ f'\ g = (\circ)\ \hat{f}'\ g$

$(\hat{\circ})\ f' = (\hat{\circ})\ (\text{bind}\ f')$

$(\hat{\circ})\ f = ((\hat{\circ}) \circ \text{bind})\ f$

tem unit?

unit : $\text{Int} \to \text{Int} \times \text{String}$

unit $x \overset{\text{def}}{\equiv} (x, "")$ -- unit = (, "")

Generalize

Monoid

$$\text{unit} : \text{Coisado}\ \omega \Rightarrow \alpha \to \alpha \times \omega$$

$$\text{unit} \overset{\text{def}}{\equiv} (,\ \text{mempty})$$

$$(\hat{\circ}) : (\beta \to \gamma \times \omega) \to (\alpha \to \beta \times \omega) \to (\alpha \to \gamma \times \omega)$$

$$f' \mathbin{\hat{\circ}} g' \overset{\text{def}}{\equiv} \hat{f}' \circ g'$$

ou: $(f' \mathbin{\hat{\circ}}) \overset{\text{def}}{\equiv} (\hat{f}' \circ)$

ou: $(\hat{\circ}) \overset{\text{def}}{\equiv} (\circ) \circ \text{bind}$

$$\text{bind} : (\alpha \xrightarrow{f'} \beta \times \omega) \to (\alpha \times \omega \xrightarrow{\hat{f}'} \beta \times \omega)$$

$$\underbrace{\text{bind}\ f'}_{\hat{f}'}\ (x, s) \overset{\text{def}}{\equiv} \text{let}\ (v_f, m_f) = f'\ x \ \text{in}\ (v_f, s \lozenge m_f)$$

Monoidal

lec26
2025-01-20

typeclass (Functor f) ⇒ Monoidal (f : Ty → Ty)

```
unit  : f ()
(**)  : f α → f β → f (α × β)

unit' : 𝟙 → f ()
(**') : (f α × f β) → f (α × β)
```

} ...

() × Int ≅ Int

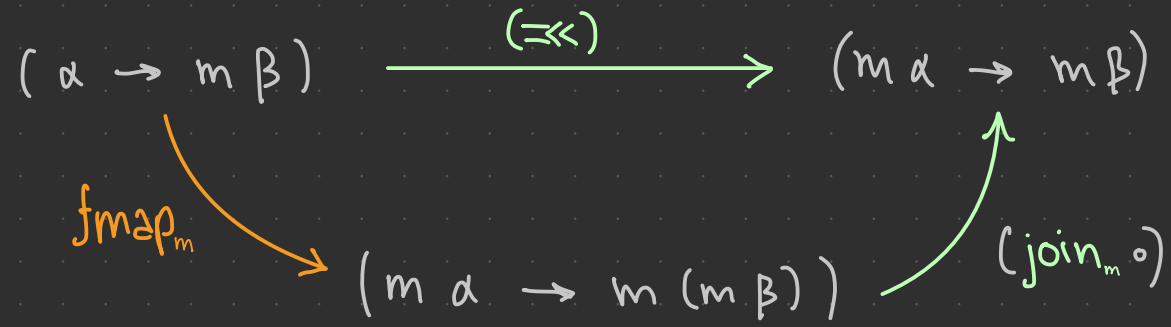
```
unit ** v "=" v

u ** unit "=" u

(u ** v) ** w = u ** (v ** w)

fmap (g × h) (u ** v) = fmap g u ** fmap h v
```

gratuito em Haskell

FAMM

fmap , join ⊢ (>>=)

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



fmap , join ⊢ (>>=)

fmap, join ⊣ (>>=), pure

Prova 3

2025-01-24

T (28 pts)

read3Ints : IO (M (Int × Int × Int))

R Escolha 6 dos 10. (6 × 12 pts)

Unit, (**) ⊣⊢ pure, (⊛)

fmap, join ⊢ (>>=)

fmap, join ⊣ (>>=), pure

Functor instance para um dos : ⟨ (δ →) | (→ γ) ⟩

Uma fmap que respeita a lei de (∘) mas não de id

sequenceAL : Applicative f ⇒ List (f α) → f (List α)

lift $\quad fmap_f : (\alpha \to \beta) \to (f\,\alpha \to f\,\beta)$

$ap_m := (\circledast_a) \equiv splat_a : a\,(\alpha \to \beta) \to (a\,\alpha \to a\,\beta)$

bind $\begin{cases} (=\!\!\ll_m) : (\alpha \to m\,\beta) \to (m\,\alpha \to m\,\beta) \\ (\gg\!\!=_m) : m\,\alpha \to (\alpha \to m\,\beta) \to m\,\beta \end{cases}$

$join_m : m\,(m\,\alpha) \to m\,\alpha$

$return_m := pure_a : \alpha \to a\,\alpha$

kleisli $\begin{cases} (\gg\!\!\Rightarrow_m) : (\alpha \to m\,\beta) \to (\beta \to m\,\gamma) \to (\alpha \to m\,\gamma) \\ (\Leftarrow\!\!\ll_m) : (\beta \to m\,\gamma) \to (\alpha \to m\,\beta) \to (\alpha \to m\,\gamma) \end{cases}$

$(\gg_m) := (*\!\!\gg_a) : a\,\alpha \to a\,\beta \to a\,\beta$

$void_f : f\,\alpha \to f\,1$

$unit_n : n\,()$ $\qquad$ $unit'_n : 1 \to n\,()$

$(**)_n : n\,\alpha \to n\,\beta \to n\,(\alpha \times \beta)$ $\qquad$ $(**')_n : (n\,\alpha \times n\,\beta) \to n\,(\alpha \times \beta)$