L2 →

Reverse: List α → List α
Reverse [] = []
Reverse (x:xs) = Reverse xs ++ [x]

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

```
reverse . reverse =  id
map f . map g     =  map (f∘g)
map f (xs ++ ys)  =
map id            =  id
filter p . map f  =
sum . map (n *)   =
```

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

map: (α → β) → List α → List β
map f [] = []
map f (x:xs) = f x : map f xs

filter: (α → Bool) → List α → List α
filter p [] = []
filter p (x:xs) = if p x then
x : filter p xs
else
filter p xs

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA $(\forall f, g : (\alpha \to \beta))\,[(\forall l : List\ \alpha)[map\ f \cdot map\ g\ l = map\ (f \circ g)\ l]]$

Sejam f, g funções de (α → β).
Indução no l.
-- Base
Calculamos:

(map f ∘ map g) [] = map f (map g []) [(∘).1]
= map f [] [map.1]
= [] [map.1]

map (f∘g) [] = [] [map.1]

-- PASSO INDUTIVO
Suponha (map f ∘ map g) xs = map (f∘g) xs. [H.i]
Calculamos:

(map f ∘ map g) (x:xs) = map f (map g (x:xs)) [(∘).1]
= map f (g x : map g xs) [map.2]
= f (g x) : map f (map g xs) [map.2]

map (f∘g) (x:xs) = (f∘g) x : map (f∘g) xs [map.2]
= f (g x) : (map f ∘ map g) xs [H.i] & (∘).1
= f (g x) : map f (map g xs) [(∘).1]
■

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | = Id ✓ | `map id` | = map (reverse . reverse) |
| `map f . map g` | = map (f . g) ✓ | `filter p . map f` | = |
| `map f (xs ++ ys)` | = map f xs ++ map f ys ✓ | `sum . map (n *)` | = sum . reverse (map (n *)) |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
reverse : Lista a → lista a
reverse [] = []
reverse (x:xs) = reverse xs : x

filter : (a → B) → lista a → Lista B
filter p [] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

```
map : (A → B) → Lista a → Lista a
map f [] = []
map f (x:xs) = f x : map f xs
```

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA (∀ f, g) (∀ l : lista a) map f . map g l = map (f . g) l

Sejam f, g funções.
Indução

[]:
calculamos:
(map f . map g) [] = map f (map g []) [0.1]
= map f [] [map.1]
= [] [map.1]
map (f . g) [] = [] [map.1] ✓

(x:xs):
suponha (map f . map g) xs = map (f . g) xs [HI]
seja x.
calculamos:
(map f . map g) (x:xs) = map f (map g (x:xs)) [0.2]
= map f (g x : map g xs) [map.2]
= f (g x) : map f (map g xs) [map.2]
= f (g x) : (map f . map g) xs) [0.2]
= f (g x) : map (f . g) xs [HI]
= f . g x : map (f . g) xs [0.2]
= (f . g) (x : xs) [0.2]

calma aí..

✓

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(56) T

(7) **T1.** Escreva a regra de inferência que corresponde à indução do tipo Tree $\alpha$.

$$\frac{\text{ind}\varphi\ (\forall a.\ \varphi\ (\text{tip}\ a))\quad (\forall l, r : \text{tree}\ \alpha)\ [\varphi\ l\ \&\ \varphi\ r \Rightarrow \varphi\ (\text{fork}\ l\ r)]}{(\forall t : \text{tree}\ \alpha)\ [\varphi\ t]}$$

(5) **T2.** Escreva **apenas nomes e tipagens** para os destrutores do tipo Tree $\alpha$.[7]

```
Leaf : tree α -> α
Branches : tree α -> (tree α x tree α)
```

(4) **T3.** Defina o que precisa para o Tree virar um Functor.[8] (O cons tem 2 destrutores: head, tail.)
DEFINIÇÃO.

```
Maptree : (α -> β) -> tree α -> tree β
Maptree f (tip a) = Tip (f a)
Maptree f (Fork l r) = Fork (Maptree f l) (Maptree f r)
```

(16) **T4.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(2×) *forks*, *tips* : Tree $\alpha$ → Nat  *flat* : Tree $\alpha$ → List $\alpha$

(5×) *search* : $\alpha$ → Tree $\alpha$ → List Path  *fetch* : Path → Tree $\alpha$ → Maybe $\alpha$.

RESPOSTA. **Não** repita as tipagens na resposta!

```
forks (Tip a) = 0
forks (Fork l r) = 1 + (forks l + forks r)

tips (Tip a) = 1
tips (Fork l r) = tips l + tips r

flat (Tip a) = [a]
flat (Fork l r) = (flat l) ++ (flat r)

search a (Tip a) =

fetch [] (Fork _) = Nothing
fetch [] (Tip a) = Just a
fetch (d:_) (Tip _) = Nothing
fetch (d:p) (Fork l r) = if (d = L)
                         then fetch p l
                         else fetch p r
```

Maybe a

Maybe (Maybe α)

use pattern matching!

Veja gabarito

[7] Versões parciais/unsafe!

[8] **Com** tipagem; e **sem** demonstrar as leis necessárias!

(24) **T5.** Demonstre **exatamente uma das:**

(16) (i) $tips = \mathsf{S} \circ forks$;

(24) (ii) $flat \circ fmap\ f = map\ f \circ flat$. $(\forall t : tree\ \alpha)\ [flat(treemap\ f\ t) = map\ f\ (flat\ t)]$

Podes considerar dados quaisquer dos teoremas da **L1.**

Viu isso não, né? :-:

DEMONSTRAÇÃO da (i i)

Seja * uma tree α. Seja f: α -> β

Indução

caso (t = Tip a):

Logo, treemap f t = Tip (f a) [treemap.1]

Logo, flat (treemap f t) = [f a] [flat.1]

Calculamos:

map f (flat t) = map f [a] [Flat.1]

= • [f a] [map.1]

Caso t = Fork l r:

Sejam l, r: tree α tais que flat (treemap f l) = map f (flat l)

e flat (treemap f r) = map f (flat r) [H.L]

calculamos:

flat (treemap f t) = flat (Fork (treemap f l) (treemap f r)) [treemap.2]

= flat (Fork (

= flat (treemap f l) ++ flat (treemap f r) [flat.2]

= (map f (flat l)) ++ (map f (flat r)) [H.1.]

= map f (flat l ++ flat r) [Lema 1]

= map f (flat (fork l r))

= map f (flat t)

[flat.2←]

(pulou as (o).1!)

Error 404

Só isso mesmo.

(56) T

(7) **T1.** Escreva a regra de inferência que corresponde à indução do tipo Tree $\alpha$.

$$\frac{(\forall a : \alpha)\,[\varphi(tip\ \alpha)] \qquad (\forall l, r : \text{TREE}\ \alpha)\,[\varphi(l) \wedge \varphi(r) \Rightarrow \varphi(fork\ l\ r)]}{(\forall t : \text{TREE}\ \alpha)\,[\varphi(t)]}.$$

(5) **T2.** Escreva **apenas nomes e tipagens** para os destrutores do tipo Tree $\alpha$.[7]

UNTIP : TREE α → α  UNFORK : TREE α → (TREE α × TREE α)

↳ quantos destruteres tem o cons?

(4) **T3.** Defina o que precisa para o Tree virar um Functor.[8]
DEFINIÇÃO.

map : (α → β) → (TREE α) → (TREE β)  TREE : TYPE → TYPE ✓
map f (TIP α) = TIP f α
map f (FORK l r) = FORK (map f l) (map f r) ✓

(16) **T4.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(2×) *forks*, *tips* : Tree $\alpha \to$ Nat  *flat* : Tree $\alpha \to$ List $\alpha$

(5×) *search* : $\alpha \to$ Tree $\alpha \to$ List Path  *fetch* : Path $\to$ Tree $\alpha \to$ Maybe $\alpha$.

↳ caminhos que encontram o dado

RESPOSTA. **Não** repita as tipagens na resposta!

```
TIPS  TIP α  = S O
TIPS  FORK l r  = S (TIPS l + TIPS r)

FORKS  TIP α  = O
FORKS  FORK l r  = S (FORK l + FORK r)

FLAT  TIP a  = [a]
FLAT  FORK l r  = (FLAT l) ++ (FLAT r)

SEARCH  n  TIP a  = IF n = a THEN [[]] ELSE !!
SEARCH  n  FORK l r  = SEARCH l ++ SEARCH r

FETCH : l _  = NOTHING ?!
FETCH l TIP a  = just a
FETCH  FORK l r  = just (FETCH l  FETCH r)
```

[7] Versões parciais/unsafe!
[8] **Com** tipagem; e **sem** demonstrar as leis necessárias!

$(\forall l, r : \text{tree}\, a)\ [\text{tips}\ l = (S \cdot \text{forks})\, l \quad \text{tips}\,(\text{fork}\, l\, r) = \text{tips}\, r = (S \cdot \text{forks})\, r \Rightarrow (S \cdot \text{forks})\,\text{fork}\, l\, r$

(24) **T5.** Demonstre **exatamente uma das:**

(16) (i) $tips = \mathsf{S} \circ forks$; $\sim \vee\ (\forall a : a)\ [\text{tips}\ \text{tip}\, a = S \circ \text{forks}\ \text{tip}\, a]$.

(24) (ii) $flat \circ fmap\ f = map\ f \circ flat$.

Podes considerar dados quaisquer dos teoremas da **L1.**

DEMONSTRAÇÃO.

I.

Indução. Base:
Seja $a : a$
Calculamos:

$\text{tips}\,(\text{tip}\, a) = S\,0 \quad [\text{tips}.1]$

$S \cdot \text{forks}\ \text{Tip}\, a = S\,(\text{forks}\ \text{tip}\, a)\ [(\cdot).1]$
$= S\,0 \quad [\text{forks}.1]$

Caso indutivo:
Sejam $l, r : \text{tree}\, a$ t.q $\text{tips}\, l = (S \circ \text{forks})\, l$ e
$\text{tips}\, r = (S \circ \text{forks})\, r$.

Calculamos:

$\text{tips}\,(\text{fork}\, l\, r) = \text{tips}\, l + \text{tips}\, r \quad [\text{tips}.2]$
$= (S \circ \text{forks})\, l + (S \circ \text{forks})\, r \quad [H.1]$.

$(S \cdot \text{forks})\,\text{fork}\, l\, r = S\,(\text{forks}\,(\text{fork}\, l\, r))\ [(\cdot).1]$
$= S\,(S\,(\text{forks}\, l + \text{fork}\, r)\ [\text{Forks}.2]$
$= (S \circ \text{forks})\, l + (S \circ \text{forks})\, r\ [\text{LEMMA}.1]$.

Só isso mesmo.

LEMMA.1 $(\forall n, m : \mathbb{NAT}) [\, S(S(n+m)) = Sn + Sm$

Sejam $n : \mathbb{NAT}$

Estranho enunciar este lemma.
Veja o gabarito!

Indução:

Base:

Calculemos:

$S(S(n+0)) = S(S(n))$ ✗ veja a def de (+)!

$Sn + S0 = S(S(n+0))$ (?)

$= S(S(n))$

Passo Indutivo:

Sup. $S(S(n+m)) = Sn + Sm$ [H.I]

Calculemos:

$S(S(n+Sm)) = S(S(S(n+m)))$ [(+).2] ~

$= S(Sn + Sm)$. [H.I] ~

$Sn + S(Sm) = S(Sn + Sm)$. [(+).2] ✓

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

```
reverse . reverse = id                  map id              = id
map f . map g     = map (f . g)         filter p . map f    =
map f (xs ++ ys)  = map f xs ++ map f ys sum . map (n *)     =
```

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
map :: (β -> γ) -> [α] -> [β]
map f [] = []
map f (x:xs) = f x : map f xs
```

é bastante suspeito o fato que tu ta usando duas notações distintas em maneira consistente. Tem um mal-entendido aqui?

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA ~~map f . map g = map (f . g)~~.

(∀f : α -> β)(∀g : β -> γ)(∀xs : List α)[(map f . map g) xs = map (f . g) xs]

Seja f : α -> β e g : β -> γ

Indução nos xs.

Base:

Calculamos:

```
(map f . map g) []  = map f (map g [])      [∘.1]
                    = map f []              [map.1]
                    = []                    [map.1]
                    = map (f ∘ g) []        [map.1 ←]
```

Grr...

Passo indutivo:

Seja xs : List α tal que (map f . map g) xs = map (f ∘ g) xs.

Calculamos:

```
(map f . map g) (x:xs) = map f (map g (x:xs))     [(∘.1)]
                       = map f (g x : map g xs)   [map.2]
                       = f (g x) : map f (map g xs)   [map.2]
                       = f (g x) : (map f . map g) xs   [(∘.1 ←]
                       = f (g x) : (map f ∘ g) xs       [H.i]
                       = map f∘g (x:xs)                 [map.2←]
```

?

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | = reverse | `map id` | = id |
| `map f . map g` | = map (f · g) | `filter p . map f` | = map f · filter (p · f) |
| `map f (xs ++ ys)` | = (map f xs) ++ (map f ys) | `sum . map (n *)` | = (n *) . sum |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
map : (α → β) → List α → List β
map f [] = []
map f (x : xs) = f x : map f xs

filter : (α → Bool) → List α → List α
filter p [] = []
filter p (x : xs) = if p x then x : filter p xs
                    else filter p xs

reverse : List α → List α
reverse [] = []
reverse (x : xs) = reverse xs ++ x
```

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA map f . map g = map (f · g) .

Indução.

Base:

Cálculos:

```
map f . map g []  :  map f (map g [])        (.).1
                  :  map f []                (map).1
                  :  []                      (map).1

map (f . g) []    :  []                      (map).1
```

Passo indutivo:

Seja $x_s$ tal que $(\text{map } f \cdot \text{map } g)\ x_s = \text{map } (f \cdot g)\ x_s$. (HI)

Seja $x : \alpha$.

Cálculos:

```
(map f . map g) (x : xs)  =  map f (map g (x : xs))      (.).1
                          =  map f (g x : map g xs)      (map).2
                          =  (f (g x)) : (map f (map g xs))   (map).2
                          =  ((f . g) x) : (map f . map g) xs)   ((.).1)
                          =  ((f . g) x) : (map (f . g) xs)      (HI)
                          =  map (f . g) (x : xs)        (map).2
```

---

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

```
reverse . reverse = id                  map id                  = id
map f . map g     = map (f . g)         filter p . map f = map f . filter (p . f
map f (xs ++ ys)  = map f xs ++ map f ys  sum . map (n *)  = (n *) . sum
```

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
map : (α → β) → List α → List β
map f [] = []
map f (x : xs) = f x : map f xs

reverse : List α → List α
reverse [] = []
reverse (x : xs) = reverse xs ++ [x]

filter : (α → Bool) → List α → List α
filter p [] = []
filter p (x : xs) = if p x
                    then x : filter p xs
                    else filter p xs
```

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA map f ∘ map g l = map (f ∘ g) l

Indução.

[]:

Calculamos:

```
map f ∘ map g []  = map f (map g [])      [∘]
                  = map f []              [map.1]
                  = []                    [map.1]
map (f ∘ g) []    = []                    [map.1]
```

(Desisto corrigir parenteses errados)

(x : xs):

Suponha map f ∘ map g xs = map (f ∘ g) xs. [HI]

Calculamos:

```
map f ∘ map g (x : xs) = map f (map g (x : xs))      [∘]
                       = map f (g x : map g xs)      [map.2]
                       = f (g x) : map f (map g xs)  [map.2]
                       = (f ∘ g) x : (map f ∘ map g) xs  [∘]
                       = (f ∘ g) x : map (f ∘ g) xs      [HI]
                       = map (f ∘ g) (x : xs)            [map.2←]
```

■

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | `=` | `map id` | `=` `id` ✓ |
| `map f . map g` | `=` `map (f . g)` ✓ | `filter p . map f` | `=` |
| `map f (xs ++ ys)` | `=` | `sum . map (n *)` | `=` ~~`product n . sum`~~ type error! |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
map :: (α -> β) -> [α] -> [β]
map f []     = []
map f (x:xs) = f x : map f xs
```
✓

```
filter :: (α -> Bool) -> [α] -> [α]
filter p []     = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs
```
✓

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA ☺ .

**Lema.** $\mathtt{map}\ f \cdot \mathtt{map}\ g = \mathtt{map}\ (f \cdot g)$

Indução.

Base.

$$
\begin{aligned}
(\mathtt{map}\ f \cdot \mathtt{map}\ g)\ [] &= \mathtt{map}\ f\ (\mathtt{map}\ g\ []) && [\text{def. composição}] \\
&= \mathtt{map}\ f\ [] && [\mathtt{map}.1] \\
&= [] && [\mathtt{map}.1] \\
&= \mathtt{map}\ (f \cdot g)\ [] && [\overleftarrow{\mathtt{map}}.1]
\end{aligned}
$$

Passo Indutivo.

Seja $xs : [\alpha]$ t.q. $(\mathtt{map}\ f \cdot \mathtt{map}\ g)\ xs = \mathtt{map}\ (f \cdot g)\ xs$. [H.I.]
Seja $x$.

$$
\begin{aligned}
(\mathtt{map}\ f \cdot \mathtt{map}\ g)\ (x : xs) &= \mathtt{map}\ f\ (\mathtt{map}\ g\ (x : xs)) && [\text{def. comp.}] \\
&= \mathtt{map}\ f\ (g\ x : \mathtt{map}\ g\ xs) && [\mathtt{map}.2] \\
&= (f \cdot g)\ x : \mathtt{map}\ f\ (\mathtt{map}\ g\ xs) && [\mathtt{map}.2] \\
&= (f \cdot g)\ x : (\mathtt{map}\ f \cdot \mathtt{map}\ g)\ xs && [\text{def. comp.}] \\
&= (f \cdot g)\ x : \mathtt{map}\ (f \cdot g)\ xs && [\text{H.I.}] \\
&= \mathtt{map}\ (f \cdot g)\ (x : xs) && [\overleftarrow{\mathtt{map}}.2]
\end{aligned}
$$

pulou (0).1

Uso de parens correto! ☺

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

robot!

```
(reverse . reverse) = id
map f . map g       = map (f . g)
map f (xs ++ ys)    = map f xs ++ map f ys

map id              = id
filter p . map f    = filter (p . f)
sum . map (n *)     = n * (sum l)
```

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
filtrn : (α → Bool) → List α → List α
filtrn _ [] = []
filtrn p (x:xs) =
  | if px then x : filtrn p xs
  | else filtrn p xs
```

```
map : (α → β) → List α → List β
map f [] = []
map f (x:xs) = (f x) : map f xs
```

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

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA $(\forall l)[(map\ f \circ map\ g)\ l = map\ (f \circ g)\ l]$.

Indução.
Base:
Calculamos:

$(map\ f \circ map\ g)[] \equiv map\ f\ (map\ g\ [])$ [(0).1]
$= map\ f\ []$ [map.1]
$= []$ [map.1]

$map\ (f \circ g)\ [] = []$ [map.1]

P.I.
Seja $xs : List\ \alpha$ tq. $(map\ f \circ map\ g)\ xs = map\ (f \circ g)\ xs$.
Seja $x : \alpha$.
Calculamos:

$(map\ f \circ map\ g)\ (x:xs) = map\ f\ (map\ g\ (x:xs))$ [(0).1]
$= map\ f\ (gx : map\ g\ xs)$ [map.2]
$= f(gx) : map\ f\ (map\ g\ xs)$ [map.2]

$map\ (f \circ g)\ (x:xs) = (f \circ g)\ x : map\ (f \circ g)\ xs$ [map.2]
$= f(gx) : map\ (f \circ g)\ xs$ [(0).1]
$= f(gx) : (map\ f \circ map\ g)\ xs$ [H.I]
$= f(gx) : map\ f\ (map\ g\ xs)$ [(0).1]

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

```
reverse . reverse =                map id          =
map f . map g     =
map f (xs ++ ys)  =                filter p . map f =
                                   sum . map (n *)  =
```

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
map :: (α → β) → List α → List β
map _ [] = []
map f (x:xs) = f x : map f xs

reverse :: List α → List α
reverse [] = []
reverse (x:xs) = concat x (reverse xs)

filter :: (α → Bool) → List α → List α
filter _ [] = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA map id = id .

(∀l : List α) [ map id l = id l ]

Indução

Base (map id [] = id [])

map id [] = [] [map.1]

id [] = [] [id.1]

Passo indutivo ((∀l : List α)[map id l = id l ⇒ (∀a : α)[map id a:l = id a:l]]

Seja l : List α tq. map id l = id l

Seja a : α.

Calculamos.

map id a:l = id a : map id l [map.2]

= a : map id l [id.1]

= a : id l [escolha de l]

= a : l [id.1]

id a:l = a:l [id.1]

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

```
reverse . reverse = id                  map id                = id
map f . map g     = map (f.g)           filter p . map f      = map f . filter (p.f)
map f (xs ++ ys)  =                     sum . map (n *)       = (n *) . sum
```

map f xs ++ map f ys

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
reverse :: List α → List α
reverse [] = []
reverse (x:xs) =
    (reverse xs) ++ [x]
```

```
map :: (α→β) → List α → List β
map f [] = []
map f (x:xs) = f x : map f xs

filter :: (α → Bool) → List α → List α
filter p [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs
```

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA map f . map g = map (f . g).

Seja xs : List α
Indução no xs
Base:
Calculemos:

$$(map\ f \cdot map\ g)\ [] = map\ f\ (map\ g\ [])\quad [(\cdot).1]$$
$$= map\ f\ []\quad [map.1]$$
$$= []\quad [map.1]$$

$$map\ (f \cdot g)\ [] = []\quad [map.1]$$

Passo Indutivo:

Seja ys : List α tal que $(map\ f \cdot map\ g)\ ys = map\ (f \cdot g)\ ys$ [H.I]
Seja y : α
Vai demonstrar que $(map\ f \cdot map\ g)\ (y:ys) = map\ (f \cdot g)\ (y:ys)$
Calculemos:

$$(map\ f \cdot map\ g)\ (y:ys) = map\ f\ (map\ g\ (y:ys))\quad [(\cdot).1]$$
$$= map\ f\ (g\ y : map\ g\ ys)\quad [map.2]$$
$$= f\ (g\ y) : map\ f\ (map\ g\ ys)\quad [map.2]$$
$$= (f \cdot g)\ y : (map\ f \cdot map\ g)\ ys\quad [(\cdot).1\leftarrow]$$
$$= (f \cdot g)\ y : map\ (f \cdot g)\ ys\quad [H.I]$$
$$= map\ (f \cdot g)\ (y:ys)\quad [map.2\leftarrow]$$

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | = map id | `map id` | = reverse ∘ reverse |
| `map f . map g` | = map(f∘g) | `filter p . map f` | = map f ∘ filter p . f |
| `map f (xs ++ ys)` | = map f xs ++ map f ys | `sum . map (n *)` | = n * sum |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
reverse : list α → list α
reverse [] = []
reverse x:xs = (reverse xs) ++ [x]
```

filter está no lemmata

```
map : (α → β) → list α → list β
map f [] = []
map f x:xs = f x : map f xs
```

if px then px : filter p xs else filter p xs

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA $(\forall xs : list\ \alpha)\ [(map\ f \circ map\ g)\ xs = map\ (f \circ g)\ xs]$

Indução.
Caso Nil:
Calculamos:

$$\begin{aligned}
(map\ f \circ map\ g)\ [\,] &= map\ f\ (map\ g\ [\,]) && [0.1]\\
&= map\ f\ [\,] && [map.1]\\
&= [\,] && [map.1]
\end{aligned}$$

$map\ (f \circ g)\ [\,] = [\,]$ $[map.1]$.

Caso Cons:
Calculamos:

$$\begin{aligned}
(map\ f \circ map\ g)\ x:xs &= map\ f\ (map\ g\ (x:xs)) && [0.1]\\
&= map\ f\ (g\,x : map\ g\ xs) && [map.2]\\
&= f\,(g\,x) : map\ f\ (map\ g\ xs) && [map.2]
\end{aligned}$$

$$\begin{aligned}
map\ (f \circ g)\ (x:xs) &= (f \circ g)\,x : map\ (f \circ g)\ xs && [map.2]\\
&= f\,(g\,x) : map\ (f \circ g)\ xs && [0.1]\\
&= f\,(g\,x) : (map\ f \circ map\ g)\ xs && [HI]\\
&= f\,(g\,x) : map\ f\ (map\ g\ xs) && [0.1]
\end{aligned}$$

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | = map id | `map id` | = reverse . reverse |
| `map f . map g` | = map (f . g) | `filter p . map f` | = map f (filter p ~~...~~) |
| `map f (xs ++ ys)` | ~~= map (concat xs ys)~~ map f (concat xs ys) | `sum . map (n *)` | = pw (+) . foldl (*) n |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
map :: (α → β) → List α → List β
map f [ ] = [ ]
map f (x:xs) = f x : map f xs
```

```
filter (α → Bool) → List α → List α
filter p [ ] = [ ]
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs
```

Reverse na lemmata!

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA filter p . map f = map f (filter p) !!

Sejam f (α → β), p (β → Bool)
Indução.
[]:
Calculemos:
filter p . map f [ ]
= filter p [ ]
= [ ]

type error!
como sumiu o . ?

(map . 1)] ([filter . 1]]

map f (filter p [ ])
= map [ ]
= [ ]

([filter . 1]]
[(map . 1)]

(X:Xs):
~~Sejam Xs~~
~~Calculemos:~~
~~filter p map f (x:xs)~~
~~= filter p (f x : map f xs) [(map f)]~~
~~= if p f x then x : filter p (map f xs)~~

Segue na Lemmata 2.2

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

escolha ruim!

LEMMATA

```
append :: α → List α → List α
append x [] = [x]
append e (x:xs) = x : append e xs
```

```
reverse :: List α → List α
reverse [] = []
reverse (x:xs) = append x (reverse xs)
```

CONTINUAÇÃO L3.

Caso p f x:

filter p . map f (x:xs)
= filter p (f x : map f xs) [map.2]
= f x : filter p (map f xs) [filter.2]
= f x : map f (filter p xs) [(H.I)]

map f (filter p (x:xs))
= map f (x : filter p xs) [filter.2]
= f x : map f (filter p xs) [map.2]

□

Caso p f x = false:

filter p . map f (x:xs)
= filter p (f x : map f xs) [map.2]
= filter p (map f xs) [filter.2]
= map f (filter p xs) [(H.I)]

map f (filter p (x:xs))
= map f (filter p xs) [filter.2]

■

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | $= \lambda id$ | `map id` | $= \lambda id$ |
| `map f . map g` | $= map\ (f \circ g)$ | `filter p . map f` | $= map\ f \circ filter\ p$ |
| `map f (xs ++ ys)` | $= map\ f\ xs\ ++\ map\ f\ ys$ | `sum . map (n *)` | $= (n *) \circ sum$ |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
Mreverse :: [a] -> [a]
Mreverse [] = []
Mreverse (x:xs) = Mreverse xs ++ [x]
```

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : map f xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) =
  | p x = x : filter p xs
  | otherwise = filter p xs
```

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA $(\forall xs : L)[map\ id\ xs = id\ xs]$.

Grr...

Indução.
Base.
calculemos:
map id []
= [map.1]
[]
= [← id.1]
id []

Passo indutivo.
Seja xs t.q. map id xs = id xs. #HIP
Seja x.
calculemos
map id (x:xs)
= [map.2]
(id x) : map id xs
= [HIP]
(id x) : id xs
= [id.1]
x : xs
= [← id.1]
id (x : xs)

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | = ~~reverse (reverse )~~ | `map id` | = id |
| `map f . map g` | = map (f∘g) | `filter p . map f` | = map f ∘ filter (p∘f) |
| `map f (xs ++ ys)` | = (map f xs) ++ (map f ys) | `sum . map (n *)` | = n * sum |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.

DEFINIÇÕES.

tipagens!

```
reverse [] = []
reverse x:xs = reverse xs ++ [x]

map f [] = []
map f x:xs = fx : map f xs

filter n [] = []
filter n x:xs = if x = n
                then x : filter xs
                else filter xs
```

era pra ser $\alpha \to Bool$

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.

DEMONSTRAÇÃO DA map f ∘ map g = map (f∘g) .

(∀ l : List Nat) [(map f ∘ map g) l = map (f∘g) l]
Por indução no l.
Caso base (l := []).
Calculamos:

| | |
|---|---|
| (map f ∘ map g) [] = map f (map g []) | [(·).1] |
| = map f [] | [map.1] |
| = [] | [map.1] |

map (f∘g) [] = []  [map.1]

Passo Indutivo (l := x : xs).
Seja xs : List Nat tq (map f ∘ map g) xs = map (f∘g) xs  [HI].
Seja x.
Demonstremos que (map f ∘ map g) (x : xs) = map (f∘g) (x : xs).
Calculamos:

| | |
|---|---|
| (map f ∘ map g) (x : xs) = map f (map g x : xs) | [(·).1] |
| = map f (g x : map g xs) | [map.2] |
| = f (g x) : map f (map g xs) | [map.2] |
| = f (g x) : (map f ∘ map g) xs | [(·).1←] |
| = f (g x) : map (f∘g) xs | [HI] |
| = (f∘g) x : map (f∘g) xs | [(·).1←] |
| = map (f∘g) (x : xs) | [map.2←] |

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(56) **T**

(7) **T1.** Escreva a regra de inferência que corresponde à indução do tipo Tree $\alpha$.

$$\frac{(\forall a:\alpha)[\varphi\,(Tip\ a)] \quad (\forall l,r:Tree\ \alpha)[\varphi\ l\ \&\ \varphi\ r \Rightarrow \varphi\,(Fork\ l\ r)]}{(\forall t:Tree\ \alpha)[\varphi\ t]}\ Ind_{Tree\ \alpha}^{\varphi}$$

(5) **T2.** Escreva **apenas nomes e tipagens** para os destrutores do tipo Tree $\alpha$.[7]

```
value     : Tree α → α
leftTree  : Tree α → Tree α
rightTree : Tree α → Tree α
```

(4) **T3.** Defina o que precisa para o Tree virar um Functor.[8]
DEFINIÇÃO. ← eu não pedi definição de Functor! :-(

Grr...

Seja mapTree : (α → β) → Tree α → Tree β. Dizemos que Tree é Functor com mapTree se e somente se:
(i) mapTree id = id
(ii) mapTree (f ∘ g) = mapTree f ∘ mapTree g

(16) **T4.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(2×) *forks*, *tips* : Tree $\alpha$ → Nat  *flat* : Tree $\alpha$ → List $\alpha$

(5×) *search* : $\alpha$ → Tree $\alpha$ → List Path  *fetch* : Path → Tree $\alpha$ → Maybe $\alpha$.

RESPOSTA. **Não** repita as tipagens na resposta!

```
flat (Tip a) = [a]
flat (Fork l r) = flat l ++ flat r

forks (Tip a) = 0
forks (Fork l r) = S(forks l + forks r)

tips (tip a) = S0
tips (Fork l r) = tips l + tips r

fetch [] _ = Nothing
fetch (d:ds) (Tip _) = Nothing
fetch (d:ds) (Fork l r) =
```

```
search a (Tip x) = []
search a (Fork l r)
  =
```

[7] Versões parciais/unsafe!
[8] **Com** tipagem; e **sem** demonstrar as leis necessárias!

(24) **T5.** Demonstre **exatamente uma das:**

(16) (i) $tips = \mathsf{S} \circ forks$;

(24) (ii) $flat \circ fmap\ f = map\ f \circ flat$.

Podes considerar dados quaisquer dos teoremas da **L1.**

DEMONSTRAÇÃO.

-- Tip

(flat ∘ fmap f) (Tip a)

= flat ((fmap f) (Tip a)) [∘]

= flat (Tip (f a)) [fmap . 4]

= [f a]. [?]

(map f ∘ flat) (Tip a)

= map f (flat (Tip a)) [∘]

= map f [a] [flat . 4]

= [f a] [?]

-- Fork

(flat ∘ fmap f) (Fork l r)

≡ flat (fmap f (Fork l r)) [∘]

= flat (Fork (fmap f l) (fmap f r)) [fmap . 2]

≡ flat (fmap f l) (fmap f r) [flat . 2]

(map f ∘ flat) (Fork l r)

≡ map f (flat (Fork l r)) [∘]

= map f (flat l ++ flat r) [flat . 2]

Só isso mesmo.

$[x_1, x_2, x_3, \ldots, x_n] \Rightarrow [fx_1, fx_2, fx_3, \ldots, fx_n]$

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

```
reverse . reverse = id                  map id = id
map f . map g     = map (f . g)         filter p . map f = map (filter p . f)
map f (xs ++ ys)  = (map f xs) ++ (map f ys)   sum . map (n *) = (n *) . sum
```

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*. map (filter p f)
DEFINIÇÕES.

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

filter p [] = []
filter p (x:xs) =
  | px = (p x) : filter p xs
  | otherwise = filter p xs

reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar. map f (map g xs)
DEMONSTRAÇÃO DA map f . map g = map (f . g) .

caso []:

map f . map g [] = map f (map g []) = map f [] = [] = map (f . g) []

caso (x:xs):

(∀xs : list α) { map f . map g xs = map (f . g) xs } ⇒ (∀x : α) (∀xs : list α) { map f . map g (x:xs) = map (f . g) (x:xs) }

suponha map f . map g (xs) = map (f . g) xs.

assim,

```
map f . map g (x:xs)
= map f (map g (x:xs))      (.)
= map f (g x : map g xs)    map.2
= f (g x) : (map f (map g xs))   map.2
= (f . g) x : ((map f . map g) xs)   (.)
= ((f . g) x) : map (f . g) xs       H.I.
= map (f . g) (x:xs)         map.2
```

= f (g x)

---

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | ≡ Reverse | `map id` | ≡ map id |
| `map f . map g` | ≡ map fg | `filter p . map f` | ≡ [illegible] map f |
| `map f (xs ++ ys)` | ≡ map f | `sum . map (n *)` | ≡ map n |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

map:

Reverse: Para todo x ∘ Reverse i ∘ 1x

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA _____.

IX

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(56) **T**

(7) **T1.** Escreva a regra de inferência que corresponde à indução do tipo Tree $\alpha$.

$$\frac{P(0)}{T\alpha(4)}$$

(5) **T2.** Escreva **apenas nomes e tipagens** para os destrutores do tipo Tree $\alpha$.[7]

(4) **T3.** Defina o que precisa para o Tree virar um Functor.[8]
DEFINIÇÃO.

(16) **T4.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(2×) $\mathit{forks}, \mathit{tips} : \mathsf{Tree}\ \alpha \to \mathsf{Nat}$ $\qquad$ $\mathit{flat} : \mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(5×) $\mathit{search} : \alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ $\qquad$ $\mathit{fetch} : \mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha.$

RESPOSTA. **Não** repita as tipagens na resposta!

IX

[7] Versões parciais/unsafe!
[8] **Com** tipagem; e **sem** demonstrar as leis necessárias!

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | = reverse² | `map id` | = map id |
| `map f . map g` | = mmph ?! | `filter p . map f` | = reverse |
| `map f (xs ++ ys)` | = map f (zs) | `sum . map (n *)` | = filter (n*) |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

reverse: função reversa
map: função mapa
filter: função de filtro

?

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA reverse . reverse = reverse²

Quando a igualdade é multiplicada por ela mesma,
sempre é apresentada por um valor ao quadrado,
ou seja (reverse²)

?!?

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

| | | | |
|---|---|---|---|
| `reverse . reverse` | ≠ Reverse(REVERSE List α) = List α | `map id` | ✓ = Id |
| `map f . map g` | ≠ map f (g (x)) | `filter p . map f` | ≠ Map f (P) |
| `map f (xs ++ ys)` | ≠ f(xs) ++ f(ys) | `sum . map (n *)` | ≠ n + Sum . map (*) |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

| | | |
|---|---|---|
| ~~Reverse: List α -> List β~~<br>Reverse: List α -> List α<br>Rev [] ys = ys<br>Rev x:xs ys = xs ++ x:ys | map: (α -> β) -> List α -> List β<br>map f [] = []<br>map f (x:xs) = if fx then x : map xs<br>else map xs | filter: (α -> β) -> List α -> List β<br>filter f [] = []<br>filter f (x:xs) = if fx then x : filter f xs<br>else filter f xs |

type error

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA Reverse α . Reverse α = List α .

Seja

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(38) **L**

(12) **L1.** Complete as igualdades seguintes **com algo interessante:**[6]

do nada!

| | | | |
|---|---|---|---|
| `reverse . reverse` | $\equiv$ id ✓ | `map id` | $\not\equiv$ (x:xs) (SEM ALTERAÇÕES) |
| `map f . map g` | $\not\equiv$ map f.g | `filter p . map f` | $\not\equiv$ filter p ((x+f)s) |
| `map f (xs ++ ys)` | $\not\equiv$ ((x+f)s ++ (y+f)s) | `sum . map (n *)` | $\not\equiv$ x*n + sum . map (n*) |

(6) **L2.** Defina recursivamente as funções: *reverse*, *map*, *filter*.
DEFINIÇÕES.

```
reverse : Listα → Listα
[ ] = [ ]
(x:xs) = (xs:x) : reverse (x':xs)

map : α → Listα → Listα
map f [ ] = [f]
map f (x:xs) = f+x : map f (xs)

filter : α → Listα → Maybe α
filter f [ ] = Nothing
filter g (x:xs) = if x=g return Just
                  else filter g (xs)
```

(20) **L3.** Escolha **exatamente uma da primeira coluna** do **L1** para demonstrar.
DEMONSTRAÇÃO DA "reverse . reverse".

Seja (x:xs) : Listα.
Calculemos:

reverse . reverse = reverse (reverse (x:xs)) (.)

= reverse (xs:x) (reverse.2)

~~= (x:xs)~~

= id. ((x:xs) novamente)

[6] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(56) **T**

(7) **T1.** Escreva a regra de inferência que corresponde à indução do tipo Tree $\alpha$.

$$\frac{(\forall x : \alpha)[P\ (Tip\ x)] \qquad (\forall t_1, t_2 : Tree\ \alpha)[(P\ t_1) \,\&\, (P\ t_2) \Rightarrow P\ (Fork\ t_1\ t_2)]}{(\forall t : Tree\ \alpha)[P\ t]}$$

✓ $P : Tree\ \alpha \to Prop$

(5) **T2.** Escreva **apenas nomes e tipagens** para os destrutores do tipo Tree $\alpha$.[7]

```
unTip : Tree α → α          unFork : Tree α → Tree α × Tree α
unTip (Tip x) = x           unFork (Fork l r) = (l, r)
```

dois destrutores!

(4) **T3.** Defina o que precisa para o Tree virar um Functor.[8]
DEFINIÇÃO.

```
fmap : (α → β) → Tree α → Tree β
fmap f (Tip x) = Tip (f x)
fmap f (Fork l r) = Fork (fmap f l) (fmap f r)
```

(16) **T4.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(2×) *forks*, *tips* : Tree $\alpha \to$ Nat  *flat* : Tree $\alpha \to$ List $\alpha$

(5×) *search* : $\alpha \to$ Tree $\alpha \to$ List Path  *fetch* : Path $\to$ Tree $\alpha \to$ Maybe $\alpha$.

RESPOSTA. **Não** repita as tipagens na resposta!

```
forks (Tip _) = 0
forks (Fork l r) = S(forks l + forks r)

tips (Tip _) = S 0
tips (Fork l r) = tips l + tips r

flat (Tip x) = [x]
flat (Fork l r) = flat l ++ flat r

search x (Tip y) = if x=y then [[]]
                          else []
search x (Fork l r) =
  map (cons L) (search x l) ++ map (cons R) (search x r)
```

!

```
fetch [] (Tip x) = Just x
fetch _ (Tip x) = Nothing
fetch [] (Fork _ _) = Nothing
fetch (x:xs) (Fork l r) =
  if x=L
  then fetch xs l
  else fetch xs r
```

↳ faça com pattern matching (veja gabarito)

[7] Versões parciais/unsafe!
[8] **Com** tipagem; e **sem** demonstrar as leis necessárias!

(24) **T5.** Demonstre **exatamente uma das:**

(16) (i) $tips = \mathsf{S} \circ forks$;

(24) (ii) $flat \circ fmap\ f = map\ f \circ flat$.

Podes considerar dados quaisquer dos teoremas da **L1.**

DEMONSTRAÇÃO.

Seja $f : \alpha \to \beta$

Indução -- Base

Seja $x : \alpha$

Calculamos:

(flat ∘ fmap f) (Tip x)
= flat (fmap f (Tip x)) [0.1]
= flat (Tip (f x)) [fmap.1]
= [f x] [flat.1]

(map f ∘ flat) (Tip x)
= map f (flat (Tip x)) [0.1]
= map f [x] [flat.1]
= f x : (map f []) [map.2]
= [f x] [map.1]

-- Passo indutivo

Sejam $t_1, t_2$ t.q.

(flat ∘ fmap f) $t_1$ =
(map f ∘ flat) $t_1$
e (flat ∘ fmap f) $t_2$ =
(map f ∘ flat) $t_2$ [H.I]

Calculamos:

(flat ∘ fmap f) (Fork $t_1$ $t_2$)
= flat (fmap f (Fork $t_1$ $t_2$)) [0.1]
= flat (Fork (fmap f $t_1$) (fmap f $t_2$)) [fmap.2]
= flat (fmap f $t_1$) ++ flat (fmap f $t_2$) [flat.2]
= (flat ∘ fmap f) $t_1$ ++ (flat ∘ fmap f) $t_2$ [0.1]

(map f ∘ flat) (Fork $t_1$ $t_2$)
= map f (flat (Fork $t_1$ $t_2$)) [0.1]
= map f (flat $t_1$ ++ flat $t_2$) [flat.2]
= map f (flat $t_1$) ++ map f (flat $t_2$) [++ law]
= (map f ∘ flat) $t_1$ ++ (map f ∘ flat) $t_2$ [0.1]
= (flat ∘ fmap f) $t_1$ ++ (flat ∘ fmap f) $t_2$ [H.I]

Só isso mesmo.