(48) D

6 (16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

0	`sum . map sum`	= SUM. MAP SUM. REVERSE
0	`filter p . map f`	= REVERSE. REVERSE. FILTER P. MAPF
2	`map f . map g`	= MAP(F.g)
2	`map f . take n`	= TAKEm. MAPF
0	`filter p . concat`	= REVERSE. FILTER P. CONCAT
0	`take m . take n`	= TAKE m
0	`concat . map reverse` (concat)	= REVERSE. CONCAT
2	`sum . map double`	= DOUBLE-SUM

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

12

```
MAP :: (a -> b) -> [a] -> [b]
MAP F [] = []
MAP F (x:xs) = F x : MAP F xs
```

DEMONSTRAÇÃO DE MAPF. MAPg = .MAP(F.g)

IREMOS PROVAR POR INDUÇÃO QUE MAP F. MAPg = MAP(F.g)

CASO BASE: (MAP F. MAPg) [] ≟ MAP(F.g) [].

CALCULAMOS: (MAPF. MAPg) [] = MAPF (MAPg []) (.).1
= MAPF ([]) (MAP.1)
= [] (MAP.1) (1.1) ✓

TEMOS TAMBÉM QUE: MAP (F.g) [] = [] (MAP.1) (1.2) ✓

LOGO, COM (1.1) E (1.2), PROVAMOS O CASO BASE.

escolha de nome ruim!

P.I: (∀x:a) (∀y:[a]) [(MAPF. MAPg) y = (MAP(F.g) y) ⇒ MAP F. MAPg (x:y) = MAP(F.g) (x:y)]

CALC.: (MAPF. MAPg) [x] = MAPF (MAPg [x]) = MAPF ([g x]) = MAPF (g x : MAPg []) (.).1 e MAP.2
= F(g x) : MAPF [] = [F(g(x))] (MAP.1) e (MAP.2) e (MAP.1)
(2.2)

MAP(F.g) [x] = (F.g) x : MAP(F.g) [] = [F(g x)] (MAP.2) e (MAP.1) e (.).1)
(2.3)

LOGO, (MAPF. MAPg) [x] = MAP(F.g) x ESTÁ PROVADO POR (2.2) E (2.3) E O P.I TAMBÉM.

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

Esse poderia ter sido uma segunda base, mas sem um passo indutivo, tu acabou demonstrando apenas para listas de tamanho 0 ou 1.

(52) F

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** Definição recursiva da *sum*.

✓

SUM :: Num a ⇒ [a] → a (Num a, Num a)

SUM (X:XS) = X + SUM XS (2)

SUM [] = 0 (1)

(28) **(ii)** Demonstração da (S).

10

INDUÇÃO

CASO BASE: SUM ([] ++ YS) $\stackrel{?}{=}$ SUM [] + SUM YS

CALC. SUM ([] ++ YS) = SUM YS ((++).1)

OUTRO LADO: SUM [] + SUM YS = SUM YS (SUM.1)

LOGO, SUM ([] ++ YS) = SUM [] + SUM YS

P.I: SUM ([x] ++ YS) $\stackrel{?}{=}$ SUM [x] + SUM YS

∀x:a

CALC.: SUM ([x] ++ YS) = SUM (x:[] ++ YS) ((++).2)

= SUM (x:YS) ((++).1)

= x + SUM YS (SUM.2) (2.2)

OUTRO LADO: SUM [x] + SUM YS = (x + SUM []) + SUM YS (SUM.2)

= (x + 0) + SUM YS (SUM.1)

= x + SUM YS (2.3)

COM 2.2 e 2.3, PROVAMOS O P.I, LOGO SUM (XS ++ YS) = SUM XS + SUM YS

(mesmo problema :))

[[1,2,3],[4,5,6]] ⟶ [[4,5,6],[1,2,3]] reverse
[[3,2,1],[6,5,4]] concat
[3,2,1,6,5,4] [4,5,6,1,2,3] reverse

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

2 `sum . map sum` = sum . concat
2 `filter p . map f` = map f . filter (p ∘ f)
2 `map f . map g` = map (f . g)
2 `map f . take n` = take n . map f
2 `filter p . concat` = concat . map (filter p)
2 `take m . take n` = take (min m n)
0 `concat . map reverse` = reverse . concat . reverse
2 `sum . map double` = double . sum

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

32!

Definições.

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

Demonstração de _____ .

(∀f: b->c)(∀g: a->b)(∀xs: List a)[(map f ∘ map g) xs = map (f ∘ g) xs]
<=>
(∀f: b->c)(∀g: a->b)(∀xs: List a)[map f (map g xs) = map (f ∘ g) xs] [def ∘]
Sejam f: b->c e g: a->b.
Indução.
Base.
Calculamos:
map f (map g []) = map f [] [map.1]
= [] [map.1]
map (f ∘ g) [] = [] [map.1]
✓
Passo indutivo.
Seja xs: List a tal que map f (map g xs) = map (f ∘ g) xs [HI]
Seja x: a
Calculamos:
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 ∘ g) xs [HI]
map (f ∘ g) (x:xs) = (f ∘ g) x : map (f ∘ g) xs [map.2]
= f (g x) : map (f ∘ g) xs [def ∘]
✓

[2] Definição. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(52) **F2.** *Aja numa maneira melhor:* **(i)** defina a *sum* como caso especial dum conceito mais abstrato capaz de definir todas as funções que o programador considerou nos exemplos acima; **(ii)** enuncie claramente o teorema que precisas demonstrar para ganhar todos esses teoremas de tal programador, de graça;[3] **(iii)** demonstre o teorema que enunciou no **(ii)**.

(6) DEFINIÇÃO LEGAL DA *sum*.

5 ✓

```
sum : Nat -> Nat -> Nat
sum = foldl (+) 0
```

por que l ?

```
foldl : (b -> a -> b) -> b -> [a] -> b
foldl op v [] = v
foldl op v (x:xs) = foldl op (v `op` x) xs
```

(8) TEOREMA.

7 ✓

Para toda op : l -> a -> l associativa, para todo v : l tal que v é identidade esquerda de op, para todas xs, ys : List a. Temos:

```
foldl op v (xs ++ ys) = foldl op v xs `op` foldl op v ys
```

(38) DEMONSTRAÇÃO.

24

Seja op : b -> a -> b tal que op é associativa

↳ onde precisou?

Seja ys : List a

Indução

Base

Seja v : b tal que v é identidade esquerda de op

Calculamos:

```
foldl op v ([] ++ ys) = foldl op v ys                    [(++).1] ✓
(foldl op v []) `op` (foldl op v ys) = v `op` foldl op v ys   [(++).1]
                                     = foldl op v ys          [esc. b]
```

Passo indutivo

Seja xs : List a tal que

```
(∀v : b) [v é b-id de op => foldl op v (xs ++ ys) = foldl op v xs `op` foldl op v ys]   HI
```

Seja v : b tal que v é id esq de op

Seja x : a

Calculamos:

→ já "na segunda chamada" do foldl esse argumento não vai ser mais a id.

```
foldl op v ((x:xs) ++ ys) = foldl op v (x:(xs ++ ys))        [(++).2]
                          = foldl op (v `op` x) (xs ++ ys)   [foldl.2]

(foldl op v (x:xs)) `op` (foldl op v ys)
                  = [foldl.2]
(foldl op (v `op` x) xs) `op` (foldl op v ys)
                  = [HI]
foldl op (v `op` x) (xs ++ ys) ✓
```

Só isso mesmo.

[3] Não esqueça incluir as hipoteses necessárias!

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
sum . map sum          = sum . concat
filter p . map f       = map f . filter p
map f . map g          = map (f . g)
map f . take n         = take n . map f
filter p . concat      = concat . map (filter p)
take m . take n        = take (m + n)
concat . map reverse   = concat . map reverse
sum . map double       = double . sum
```

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

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

DEMONSTRAÇÃO DE map f . map g

(∀ xs :: [a]) [(map f . map g) xs = map (f . g) xs]

Indução:

Base:

Calculamos:

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

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

= [] [(map).1]

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

Passo Indutivo:

Seja ys tq (map f . map g) ys = map (f . g) ys H.I.

Seja y:ys

Calculamos:

(map f . map g) (y:ys) = map f (g y : map g ys) [(.)-def]

= f (g y) : map f (map g ys) [(map).2]

∴ (f . g) y : (map f . map g) ys [(map).2]

= (f . g) y : map (f . g) ys [H.I.]

= map (f . g) (y:ys) [(map).2]

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

```
sum :: [Nat] → Nat
sum [] = 0
sum (x : xs) = x + sum xs
```

(28) **(ii)** DEMONSTRAÇÃO DA (S).

29

(∀ xs, ys : List Nat) [sum (xs ++ ys) = sum xs + sum ys]

Indução.

Base:

Seja ys : List Nat

Calculemos:

sum ([] ++ ys) = sum ys [(++).1]

sum [] + sum ys = 0 + sum ys [(sum).1]

= sum ys + 0 [(+)-com]

= sum ys [(+).1 m := sum ys]

Passo Indutivo:

Seja ls : List Nat tal que (∀ ys) [sum (ls ++ ys) = sum ls + sum ys] HI

Seja n : Nat

Calculemos:

sum ((n : ls) ++ ys) = sum (n : (ls ++ ys)) [(++).2]

= n + (sum (ls ++ ys)) [(sum).2 x := n xs := (ls ++ ys)]

= n + (sum ls + sum ys) [H.I]

sum (n : ls) + sum ys = (n + sum ls) + sum ys [(sum).2 x := n xs := ls]

= n + (sum ls + sum ys) [(+)-assoc]

nenhum xs faz sentido aqui

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
sum . map sum          = sum xs * length xs
filter p . map f       = pick f p xs
map f . map g          = map (f . g) xs
map f . take n         = take n (map f xs)
filter p . concat      =
take m . take n        = take (m + n) xs
concat . map reverse   =
sum . map double       = sum xs + sum xs
```

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

```
sum :: List Nat → Nat
sum [] = 0
sum (x:xs) = x + sum xs
```

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

```
double :: List Nat → List Nat
double [] = []
double (x:xs) = s(s x) : double xs
```

Nat → Nat

```
double :: Num α ⇒ α → α
double x = x + x
```

DEMONSTRAÇÃO DE sum . map double = sum xs + sum xs

26

$(\forall ls : \text{List Nat})[\text{sum}(\text{map double } ls) = \text{sum } ls + \text{sum } ls]$

Indução

Passo base (sum (map double []) = sum [] + sum []

Calculamos

sum (map double []) = sum [] [map.1]

= 0 [sum.1]

sum [] + sum [] = 0 + 0 [sum.1]

= 0 [+.1]

Passo Indutivo $((\forall xs : \text{List Nat})[\text{sum}(\text{map double } xs) =$
sum xs + sum xs $\Rightarrow (\forall a : \text{Nat})[\text{sum}(\text{map double } a:xs$
= sum a:xs + sum a:xs

Continua na ultima folha...

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+\ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+\ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

List Nat → Nat

```
Sum :: Num α ⇒ [α] → α
Sum [] = 0
Sum (x:xs) = x + (Sum xs)
```

(28) **(ii)** DEMONSTRAÇÃO DA (S).

(28)

$(\forall_{xs} : List\ \alpha)(\forall_{ys} : List\ \alpha)\ [\ Sum\ (xs +\!\!+\ ys) = Sum\ xs + Sum\ ys\ ]$

~~Sejam xs: List α~~
~~Produção~~
~~Passo base Sum (xs ++ []) = Sum xs + Sum []~~
~~Sum xs (xs ++ [])~~

✓ Indução.
✓ Passo base $((\forall_{ys} : List\ \alpha)[\ Sum\ ([] +\!\!+\ ys) = Sum\ [] + Sum\ ys\ ])$:
✓ Seja $ys : List\ \alpha$.
✓ Calculemos

Sum ([] ++ ys) = Sum ys [(++).1 ys := ys] ✓
Sum [] + Sum ys = 0 + Sum ys [Sum.1]
= Sum ys + 0 [(+)-com] ✓
= Sum ys [(+)-IdR] ✓

nenhum sentido escrever isso!

Sum ys = Sum ys

Passo indutivo $((\forall_{xs} : List\ \alpha)[(\forall_{ys} : List\ \alpha)[\ Sum\ (xs +\!\!+\ ys) =$
Sum xs + Sum ys ⇒ $(\forall_{\alpha} : \alpha)[\ Sum\ (\alpha : xs +\!\!+\ ys) =$
Sum α:xs + Sum ys]]]
Seja x : List α
Seja y : List α → ?!?

continua no ultimo folha...

F1(ii) (continuação):

Suponha Sum (xs ++ ys) = Sum xs + Sum ys (1)

Seja a : α

'x' teria sido um nome muito melhor!

Calculamos

Sum (a : xs) + Sum ys = (a + Sum xs) + Sum ys [Sum.2 (x:xs) := (a:xs)]

= a + (Sum xs + Sum ys) [(+)-assoc]

= a + (Sum (xs ++ ys)) [pela hipótese (1)]

= Sum (a : xs ++ ys) [Sum.2 ← (x:xs) := (a : xs ++ ys)]

~~Sum (a : xs ++ ys) ≠ Sum (a : xs ++ ys)~~

D2 (continuação)

Seja xs : list Nat t.q. Num (map double xs) = Num xs + Num xs

Seja a : Nat.

Calculamos

Num a : xs + Num a : xs = (a + Num xs) + (a + Num xs)

[Num. 2 (x:xs) := (a:xs)]

= a + a + (Num xs + Num xs) [(+)-com/assoc]

= a + a + Num (map double xs) [pela escolha de xs]

= Num (double a) : (map double xs)

[double.1 ←]

= Num (map double a : xs) [map.2 ← f := double (x:xs) := (a:xs)]

HI

(48) D

não mexa com esse lado!

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
(sum . map sum) xs      =  sum xs
filter p . map f        =  map f . filter p
map f . map g           =  map (f . g)
map f . take n          =  take n . map f
filter p . concat       =
take m . take n         =  take (min n m)
concat . map reverse    =
sum . map double        =  (double sum . map)
```

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

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

32!

DEMONSTRAÇÃO DE ☆.

(∀f : α → β)(∀g : β → γ)(∀xs : List α)[(map f ∘ map g) xs = map (f ∘ g) xs]
[map f (map g xs) = map (f ∘ g) xs]

Seja f : α → β.
Seja g : β → γ.
Indução.
Base:

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

Passo indutivo: Seja xs : List α tal que map f (map g xs) = map (f ∘ g) xs
Seja x : α

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

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+\ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+\ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

4

```
sum :: List Nat → Nat
sum [] = 0
sum (x:xs) = x + sum xs
```

(28) **(ii)** DEMONSTRAÇÃO DA (S).

28!

```
(∀xs)(∀ys)[sum (xs ++ ys) = sum xs + sum ys]
Seja ys : List Nat
Indução.
Base:
sum ([] ++ ys) = sum ys [++.1]
sum [] + sum ys = 0 + sum ys [sum.1]
                = sum ys + 0 ((+)-com)
                = sum ys (+.1)
Passo indutivo:
Seja xs : List Nat tal que sum (xs ++ ys) = sum xs + sum ys [HI]
Seja x : nat.
sum ((x:xs) ++ ys) = sum (x : (xs ++ ys)) [++.2]
                   = x + sum (xs ++ ys) [sum.2]
                   = x + (sum xs + sum ys) [HI]
                   = (x + sum xs) + sum ys [(+)-ass]
                   = sum (x:xs) + sum ys [sum.2]
```

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

2 `sum . map sum` = sum . concat
`filter p . map f` =
2 `map f . map g` = map (f . g)
2 `map f . take n` = take n . map f
2 `filter p . concat` = concat . map (filter p)
2 `take m . take n` = take (min m n)
2 `concat . map reverse` = concat . concat
2 `sum . map double` = double . sum

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

```
sum :: [Nat] → Nat
sum [] = 0
sum (x:xs) = x + sum xs

double :: Nat → Nat
double x = SSO * x
```

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

32

DEMONSTRAÇÃO DE sum . map double = double . sum

Vou mostrar que sum (map double xs) = double (sum xs), para todo xs : [Nat]. ((.) . def)?

aqui tu já aplicou duas (.).1

Indução no xs.

Base: Calculamos:

```
sum (map double []) = sum []        (map . 1)
                    = 0             (sum . 1)
                    = SSO * 0       ((*) . 1←)
                    = double 0      (double . 1←)
                    = double (sum []) (sum . 1←)
```

Passo Indutivo: Seja xs : [Nat] tal que sum (map double xs) = double (sum xs). (H.I)
Seja x : Nat. Calculamos:

LHS

```
sum (map double (x:xs))
= sum (double x : map double xs)   (map . 2)
= double x + sum (map double xs)   (sum . 2)
= double x + double (sum xs)       (H.I)
= SSO * x + SSO * (sum xs)         (double . 1)
```

```
double (sum (x:xs)) = double (x + sum xs)
= SSO * (x + sum xs)   (double . 1)   (sum . 1)
= SSO * x + SSO * (sum xs)   (*-distributividade)
```

↑ RHS

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

Feito na D2.

(28) **(ii)** DEMONSTRAÇÃO DA (S).

26

Vou mostrar que $(\forall xs : [Nat])(\forall ys : [Nat])[\ rsum(xs ++ ys) = rsum\ xs + rsum\ ys\ ]$

Seja $ys : [Nat]$.

Indução no xs.

Base: Calculamos:

$rsum\ ([\,] ++ ys) = rsum\ ys$ $((++).1)$ ✓

$\equiv 0 + rsum\ ys$ $((+).1 \leftarrow)$

$= rsum\ [\,] + rsum\ ys$ $(rsum.1 \leftarrow)$

olha n2 (+).1 → precisa (+)_com para chegar aqui

Passo indutivo:

Seja $xs : [Nat]$ tal que $rsum(xs ++ ys) = rsum\ xs + rsum\ ys$. (I.H.)

Seja $x : Nat$.

Calculamos:

$rsum((x : xs) ++ ys) = rsum\ (x : (xs ++ ys))$ $((++).2)$

$= x + rsum(xs ++ ys)$ $(rsum.2)$

$= x + (rsum\ xs + rsum\ ys)$ $(I.H.)$

$= (x + rsum\ xs) + rsum\ ys$ $((+)\text{-assoc})$ ✓

$= rsum\ (x : xs) + rsum\ ys$ $(rsum.2 \leftarrow)$

✓

(48) **D**

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
sum . map sum          =
filter p . map f       =
map f . map g          =
map f . take n         =
filter p . concat      =
take m . take n        = take m (take (m+n))
concat . map reverse   =   (concat)
sum . map double       =
```

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

DEMONSTRAÇÃO DE take m . take n

2

$(\forall xs : [a] \mid [\forall m : a)(\forall n : a)\ [take\ m\ take\ n)(xs) = take\ m\ (take\ (m+n)\ xs)]$

Por indução

Caso m = 0

- take 0 (take n xs)
  = take n xs [take.1] ?!
  Veja a take.1 mesmo

HI: take m (take n xs) = take m (take (m+n) xs)

?

- take 0 (take (0+n) xs)
  = take 0 (take n xs) [(+).1]
  = take n xs [take.1]

Caso m = Sm'

- take Sm' (take n (x:xs))
  = take m' (x : take n xs) [take 3] ?!?

- take Sm' (take (Sm' + n) xs)
  = take Sm' (take S(m'+n) xs) [(+).2]
  = take Sm' (x : take (m'+n) xs) [take 3]
  = take m' (take (m'+n) xs) [take 3]

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

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
sum . map sum          =  foldl (\acc -> \xs -> acc + sum xs) 0
filter p . map f       =
map f . map g          =  map (f . g)
map f . take n         =
filter p . concat      =
take m . take n        =  fromIntToEnum m n
concat . map reverse   =
sum . map double       =  foldl (\acc -> \x -> acc + double x) 0
```

2

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

```
map f [] = []                (1)
map f (x:xs) = f x : map f xs (2)
```

22

DEMONSTRAÇÃO DE `map (f.g)` .

Alvo

⊢ map f . map g = map (f . g)

Demonstrando por indução.

caso base: [] Alvo map f . map g [] = map (f.g) []

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

!! def de (1) map f [] [map 1] = [] [map 1]

[] [map 1] =

[reflexividade] ■ isso não faz sentido

Continua na ultima página!

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

List Nat → Nat

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

4-

```
Sum :: [a] -> Integer
Sum [] = 0
Sum (x:xs) = x + Sum xs
```

(28) **(ii)** DEMONSTRAÇÃO DA (S).

Não escrevemos essas coisas numa demonstração

10

Alvo

⊢ Sum (xs ++ ys) = Sum xs + Sum ys

Por Indução

Caso Base: [], Seja ys: [a]

Alvo

⊢ Sum([] ++ ys) = Sum [] + Sum ys ✓

~~Sum xs~~

[(++) 1] Sum ys = 0 + Sum ys [Sum 1]

Sum ys = Sum ys

[reflexividade] ■ Grr...

Caso Indutivo

(∀xs:[a], ys[a]) [Sum (xs ++ ys) = Sum xs + Sum ys

⇒ (∀x:a, y:a) [Sum ((x:xs) ++ (y:ys)) =

Sum (x:xs) + Sum (y:ys)]

ta fazendo indução em qual argumento?

Hipotese Indutiva

Sum (xs ++ ys) = Sum xs + Sum ys ✓

map f . map g = map (f . g)

Rascunho

(∀xs:[a]) [map f . map g xs = map (f . g) xs
⇒ (∀x:a) [map f . map g x:xs = map (f . g) x:xs]]

H.I. = map f . map g xs = map (f . g) xs

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

[(.) 1]
[H.I.]

f (g x) : map (f . g) xs = f (g x) : map (f . g) xs

[refl]
Demonstração!!

D2

Continuação

Caso Indutivo:

Alvo

(∀xs:[a]) [map f . map g xs = map (f . g) xs
⇒ (∀x:a) [map f . map g (x:xs) = map (f . g) (x:xs)]]

Hipótese Indutiva

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

Passo Indutivo parenteses!

!! (map f . map g) (x:xs)

[map 2] map f (g x : map g xs)

[map 2] f (g x) : map f (map g xs)

[(.) 1 e H.I.] f (g x) : map (f . g) xs = f (g x) : map (f . g) xs [(.) 1]

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

[reflexividade] ?

parece rascunho & pulou muitos passos

(48) **D**

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

14

2 `sum . map sum` = sum . concat
0 `filter p . map f` = map f . filter p
2 `map f . map g` = map (f . g)
2 `map f . take n` = take n . map f
2 `filter p . concat` = concat . map (filter p)
2 `take m . take n` = take (min m n)
2 `concat . map reverse` = concat . concat (concat)
2 `sum . map double` = double . sum

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

32

DEFINIÇÕES.

```
sum :: [Nat] → Nat
sum [] = 0
sum (x:xs) = x + sum xs
```

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

```
double :: Nat → Nat
double x = 2 * x
```

DEMONSTRAÇÃO DE sum . map double = double . sum

(∀xs :: [Nat]) (sum . map double = double . sum)

Pela extensão da def. de composição temos que demonstrar agora sum (map double xs) = double (sum xs).

Por indução em xs.

BASE: Se xs = [] então,

sum (map double []) = sum [] [def de map] = 0 [def de sum]

Como double 0 = 0 e sum [] = 0, double (sum []) = 0. E, portanto, sum (map double []) = double (sum []).

Parece que sequer tinha algo sobre o map double [] e tu está "portantoando" sobre ele.

PI: Se xs = (x:xs) e assumindo (satisfeita por quais x,xs?) sum (map double xs) = double (sum xs) [HI], então,

Cuidado, essa é uma igualdade e não o que tu ta querendo aqui!

```
sum (map double (x:xs)) = sum (double x : map double xs)        [def map]
                        = double x + sum (map double xs)        [def. sum]
                        = double x + double (sum xs)            [HI]
                        = double (x + sum xs)                   [LEMMA1]
                        = double (sum (x:xs))                   [def sum]
```

→ isso ficou muito confuso

O "temos" sozinho dá impressão que tens nos dados.

Veja bem a diferença entre "se_, então_." e "como_, logo_."

Aqui tu precisa nenhum dos dois. Considere que neste escopo sequer tens 'xs' visível.

use map.2, sum.2, etc.

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** Definição recursiva da *sum*.

4

```
sum :: [Nat] → Nat
sum [] = 0
sum (x:xs) = x + sum xs
```

(28) **(ii)** Demonstração da (S).

26

(∀xs,ys :: [Nat]) (sum (xs ++ ys) = sum xs + sum ys)

Por indução em xs. ← de novo

Base: Se xs = [] então

sum ([] ++ ys) = ~~sum ys~~

= sum ys [def. (++)] ✓

= sum ys + 0 [def (+)] ✓

= 0 + sum ys [com. (+)] ✓

= sum [] + sum ys [def. sum] ✓

PI. Se xs = (x:xs) ~~então~~ e assumindo sum ((x:xs) ++ ys) = sum xs + sum ys [HI].

Por indução em ys. → tu acabou nunca precisando tua [HI2]. O que significa isso?

BASE1: Se ys = [] então

sum ((x:xs) ++ []) = sum (x:xs) [def (++)] ← nope!!

= sum (x:xs) + 0 [def (+)]

= sum (x:xs) + sum [] [def sum]

abuso de gerundio é crime!

PI2: Se ys = (y:ys) e assumindo sum ((x:xs) ++ ys) = sum (x:xs) + sum ys [HI2]

sum ((x:xs) ++ (y:ys)) = sum (x : (xs ++ (y:ys))) [def (++)]

= x + sum (xs ++ (y:ys)) [def sum]

= x + (sum xs + sum (y:ys)) [HI]

pulou assoc. = sum (x:xs) + sum (y:ys) [def sum]

não coma essas parenteses. Sem elas é fácil roubar

~~$(\forall xs :: [Nat])($~~

$(\forall x, y :: Nat)(\ double\ (x+y) = double\ x + double\ y\ )$ [LEMMA 1]

$double\ (x+y) = 2*(x+y)$ [def double]

$= 2*x + 2*y$ [distr (*)]

$= double\ x + double\ y$ [def double]

(F) $sum :: [Nat] \to Nat$

$sum\ [] = 0$

$sum\ (x:xs) = x + sum\ xs$

(5) $sum\ (xs ++ ys) = sum\ xs + sum\ ys$

$(\forall xs, ys :: [Nat])\ [sum\ (xs ++ ys) = sum\ xs + sum\ ys]$

Por indução em xs.

Base: Se $xs = []$ temos que provar que $sum\ ([] ++ ys) = sum\ [] + sum\ ys$

$sum\ ([] ++ ys) = sum\ ys$ [def (++)]

$= 0 + sum\ ys$

$= sum\ ys + 0$ [def (+)]

$= sum\ [] + sum\ ys$ [com (+)] [def sum]

PI: Se $xs = (x:xs)$ temos que demonstrar $sum\ ((x:xs) ++ ys) = sum\ (x:xs) + sum\ ys$.

Assumindo que $sum\ (xs ++ ys) = sum\ xs + sum\ ys$. [HI$_1$]

Por indução em ys.

BASE 2. Se $ys = []$

$sum\ ((x:xs) ++ []) = sum\ (x:xs)$

$= sum\ (x:xs) + 0$

$= sum\ (x:xs) + sum\ []$.

PI 2. Se $ys = (y:ys)$ e assumindo que $sum\ ((x:xs) ++ ys) = sum\ (x:xs) + sum\ ys$. [HI$_2$]

$sum\ ((x:xs) ++ (y:ys)) = sum\ (x : (xs ++ (y:ys)))$

$= x + sum\ (xs ++ (y:ys))$

$= x + sum\ xs + sum\ (y:ys)$ [HI$_1$]

$= sum\ (x:xs) + sum\ (y:ys)$

D2. take m ∘ take n = ??

$(\forall m : Nat, \forall n : Nat, \forall x : [a])[(take\ m \circ take\ n)\ x_n = take\ m\ (take\ n\ x_n)]$

Aqui tu tá tentando demonstrar algo imediato pela (∘).1 $f := take\ m$, $g := take\ n$

Sejam m, n : Nat. Calculemos:

base:

(take 0 ∘ take 0) x_n = take 0 (take 0 x_n) {def. de comp.}

= [] take 0 []

= take 0 []

= []

= []

Tu tá tentando "induzir" simultaneamente nos n & m, e no teu P.I. parece até incluído o xs na indução.

Nil:

(take n ∘ take m) [] = take m (take n []) (DEF. COMP.)

= [take 0 []]

= take m []

= []

= []

Passo Ind. $\forall m$ (m)

$(\forall n : Nat, m : Nat, \forall x_n : [a])$ [(take m ∘ take n) x_n = take m (take n x_n) ⇒ (take Sm ∘ take Sm) x_n = take Sm (take Sm x_n)] (H.I.)

#x : Nat

Demonst.

(take Sm ∘ take Sm) x_n = take Sm (take Sm x_n) (DEF (∘))

compare com

= take Sm (x : take n x_n) [take 3 m := Sm, x_n := x_n]

= take Sm (x : ([] ++ take m x_n)) [(++).1]

= take Sm ((x:[]) ++ take n x_n) [(++).2]

= x : (take m (take n x_n)) [take.3]

= x : ([] ++ take m (take n x_n)) [++.1]

= (x:[]) ++ take m (take n x_n) [++.2]

= [x] ++ take m (take n x_n)

=

take Sm [x : take n x_n)

x : (take m ∘ take n) x_n)

x : (take m ∘ take n) x_n)

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** Definição recursiva da *sum*.

4

sum :: [Nat] → Nat
sum [] = 0
sum (x : xs) = x + sum xs ✓

(28) **(ii)** Demonstração da (S).

?

(∀ xs : [Nat], ∀ ys : [Nat]) [sum (xs ++ ys) = sum xs + sum ys]

Por indução

Base:

Seja xs : [Nat] ;

sum [] + sum ys = sum ys [(++).1] (LHS)

sum [] ++ sum ys = sum ys [(++).1]? (RHS)

20

Análogo para ys = [].

? → tem mesmo algo mais pra ser feito aqui?

P.I.

(∀ xs : [Nat], ∀ x : Nat) [sum (xs ++ ys) = sum xs + sum ys ⇒ sum ((x : xs) ++ ys) = sum (x : xs) + sum ys]

Demonst.:

sum ((x : xs) ++ ys) = sum (x : (xs ++ ys)) [(++).2] ✓

= x + sum (xs ++ ys) [sum.2] ✓

= x + (sum xs + sum ys) [HI] ✓

= sum (x : xs) + sum ys [sum.2] ✓

pelo assoc. 6

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
sum . map sum          = sum
filter p . map f       = pick p f
1 map f . map g        = map (f g)
2 map f . take n       = take n . map f
  filter p . concat    =
0 take m . take n      = take (m+m)
  concat . map reverse =
1 sum . map double     = double (sum)
```

cuidado, tá confundindo composição com aplicação aqui!

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

DEMONSTRAÇÃO DE take m . take n = take (m+n)

8

Por indução no ~~n~~.
Seja m : Nat.
Calculamos: take m . take 0 =

Seja m, n : Nat.
Por indução no xs. cadê tal xs?

[Base]
Calculamos:
(take m . take n) [] = take m (take n []) (take.1)
= take m [] (take.2)
= [] (take.2)

take (n+m) [] = [] (take.2)

[PI]
Seja xs : List α, tal que (take m . take n) xs = take (m+n) xs (HI)
Seja x : α.
Calculamos:
~~(take m . take n) (x:xs) = take m (take n (x:xs))~~
= take

= ((.).1)

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

(52) **F2.** *Aja numa maneira melhor:* **(i)** defina a *sum* como caso especial dum conceito mais abstrato capaz de definir todas as funções que o programador considerou nos exemplos acima; **(ii)** enuncie claramente o teorema que precisas demonstrar para ganhar todos esses teoremas de tal programador, de graça;[3] **(iii)** demonstre o teorema que enunciou no **(ii)**.

(6) DEFINIÇÃO LEGAL DA *sum*.

6

```
fold :: (α -> β -> β) -> β -> [α] -> β
fold op id []     = id
fold op id (x:xs) = x `op` fold op id xs

sum :: List Nat -> Nat
sum xs = fold (+) 0 xs
```

tu tá misturando infixe/prefixe em toda a F2

(8) TEOREMA.

5

Para quaisquer xs, ys : Lista, temos que o seguinte proposição é válida: fold op (xs ++ ys) = (fold op xs) `op` (fold op ys)

tudo isso é lixo/"noise" ruim!

e.g. if (x == true) == true; else if (x == true) == false; ...

tu tá subaplicando a fold aqui

op não declarado

(38) DEMONSTRAÇÃO.

32

Por indução no xs.

[Base]

Seja ys : Lista.

Calculamos: fold op ([] ++ ys) = fold op ys [Pela (++).2]

(fold op []) `op` (fold op ys) = id `op` (fold op ys) [Pela (fold).1]

= fold op ys [Pela (id L)]

o que exatamente é isso? Cadê?

[PI]

Sejam xs, ys : Lista, t.q. fold op (xs ++ ys) = (fold op xs) `op` (fold op ys)

Seja x : α.

Calculamos: fold op (x : xs ++ ys) = fold op (x : (xs ++ ys)) [Pela (++).2]

= x `op` fold op (xs ++ ys) [Pela (fold).2]

= x `op` ((fold op xs) op (fold op ys)) [HI]

= x op (fold op xs) op (fold op ys) [Pela (fold).2←]

(fold op x:xs) op (fold op ys) = x `op` (fold op xs) `op` (fold op ys) [Pela (fold).2]

ternário?!?

como sumiram esses parenteses?

Só isso mesmo.

[3] Não esqueça incluir as hipoteses necessárias!

(48) **D**

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

2 `sum . map sum` = sum . concat
2 `filter p . map f` = map f . filter (p.f)
2 `map f . map g` = map (f.g)
2 `map f . take n` = take n . map f
`filter p . concat` = concat . filter (p.concat)
2 `take m . take n` = take (min m n).
`concat . map reverse` = concat . map concat =
2 `sum . map double` = double . sum

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

30

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

DEMONSTRAÇÃO DE map f . map g.

(∀ f,g : a->b), (∀ xs : Lista a) [map f . map g = map (f.g)]
Indução sobre xs,
Caso [];
(map f . map g) []
= map f (map g [])
= map f []
= []
[map (f.g) []
= [].
[map.1].
[map.1]
[map.1]
Caso (x:xs) : (∀ xs) [map f . map g = map (f.g)]. HIP
(map f . map g) (x:xs)
= map f (map g (x:xs))
= map f (g x : map g xs)
= f (g x) : map f (map g xs)
= (f.g) x : (map f . map g) xs
= (f.g) x : map (f.g) xs
= map (f.g) (x:xs) ∴
[(.).1]
[map.2]
[map.2]
[(.).1]
[H.I.]
[map.2]

map (f.g) (x:xs)
= (f.g) x : map (f.g) xs
[H.I.]
= (f.g) x : map f . map g xs
[(.).1]
= f (g x) : map f (map g xs)
[map.2]
= map f (g x : map g xs)
[map.2]
= map f (map g (x:xs))
[(.).1] = map f . map g (x:xs) ∴

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

necessárias!

parece com um "therefore" (mal-usado)

o que é tudo isso?!? (veja F1.ii)

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

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

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+\ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

4⁻

```
sum :: [Nat] -> Nat
sum [] = 0
sum (x:xs) = x + sum xs.
```

✓

(28) **(ii)** DEMONSTRAÇÃO DA (S).

26

(∀ xs:lista), (∀ ys:Lista) [Sum (xs ++ ys) = Sum xs + Sum ys.]

Indução sobre xs.

Caso [ ] :

Sum ([ ] ++ ys)
= Sum (ys) .·. [(++) 1]

{ Sum [ ] + sum ys
= 0 + sum ys [sum.1].
= sum ys [(+) 1].

∴

Caso (x:xs)

(∀ (x:xs):lista), (∀ ys:Lista)
→ [Sum (xs ++ ys) = Sum xs + Sum ys] (PIF).

⇒ Sum ((x:xs) ++ ys)
= Sum (x : (xs ++ ys)) - [(++) 2]
= x + Sum (xs ++ ys) - [Sum.2]
= x + (Sum xs + Sum ys) - [HI]
= (x + Sum xs) + Sum ys - [(+)-Ass.]
= Sum (x:xs) + Sum ys - [Sum.2]

Teu alvo é A = B.
Tu começou no A,
e, calculando, chegou no B.
Não tem algo mais pra ser feito!!
(Se tu tivesse chegado num terceiro M, faltaria mesmo estabelecer M = B, que poderia fazer começando pelo B para chegar no M.)

⇒ Sum (x:xs) + Sum ys - [Sum 2]
= x + Sum xs + Sum ys - [(+) Ass.]
= x + (Sum xs + Sum ys) - [+II]
= x + (Sum (xs ++ ys)) - [Sum 2]
= Sum x:(xs ++ ys) - [Sum 2]
= Sum ((x:xs) ++ ys) .·. - [(++) 2]

(48) **D**

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

2 `sum . map sum` = bsum . concat
2 `filter p . map f` = filter (P.f)
2 `map f . map g` = map (f.g)
2 `map f . take n` = take n . map f
2 `filter p . concat` = concat . map (filter P)
2 `take m . take n` = take (min m n)
`concat . map reverse` = reverse . concat . reverse
2 `sum . map double` = double . bsum

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

3| DEFINIÇÕES.

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

DEMONSTRAÇÃO DE map f . map g = map (f.g)

bizarro!

Seja xs : [a], tal que xs = [].
Seja g : a -> b
Seja f : b -> c
Calculemos:
(map f . map g) xs = map f (map g xs) (.)1
= map f [] (map 1)
= [] (map 1)
= map (f.g) [] (map 1) ←*
= map (f.g) xs (xs = [])

Seja X : a. Calculemos:
(map f . map g) (x:xs) = map f (map g (x:xs)) (.)1
= map f (g x : (map g xs)) (map g)
= f (g x) : (map f (map g xs)) (map f)
= (f.g) x : (map f (map g xs)) (.)↑
= (f.g) x : ((map f . map g) xs) (.)↑
= (f.g) x : (map (f.g) xs)
= map (f.g) (x:xs) (map ↑)

Seja g : a -> b. Seja f : b -> c. ?
Seja xs : [a], T.q. (map f . map g) xs = map (f.g) xs. ✓

necessário

*utilizando a igualdade da direita para a esquerda

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

```
Num a => [a] -> a   [Nat] -> Nat
sum [] = 0
sum (x:xs) = x + (sum xs)
```

(28) **(ii)** DEMONSTRAÇÃO DA (S).

27

de novo

Seja xs : ~~Num a => [a]~~ [Nat] tal que xs = []. Seja ys : ~~Num a => [a]~~ [Nat]

calculemos:

```
sum xs + sum ys = 0 + sum ys   (sum 1)
                = sum ys + 0   ~~(+) abelian~~ ((+) com)
                = sum ys       (+) 1)
                = sum ([] ++ ys)   ((++) 1←)
                = sum (xs ++ ys)
```

Seja ys : [Nat]. Seja xs : [Nat] t.q. sum (xs ++ ys) = sum xs + sum ys. (HI)

Seja x : Nat. calculemos:

```
sum (x:xs) + sum ys = (x + sum xs) + sum ys   (sum 2)
                    = x + (sum xs + sum ys)   ((+)-assoc)
                    = x + sum (xs ++ ys)      [H.I.]!!
                    = sum (x : (xs ++ ys))    (sum 2←)
                    = sum ((x:xs) ++ ys)      ((++) 2←)
```

(48) **D**

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

2 `sum . map sum` = sum . concat
2 `filter p . map f` = map f . filter (p . f)
2 `map f . map g` = map (f . g)
2 `map f . take n` = take n . map f
2 `filter p . concat` = concat . map (filter p)
2 `take m . take n` = take (min m n)
`concat . map reverse` = reverse . concat (concat)
2 `sum . map double` = double . sum

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

32

* A sum ou defini na F1 →

DEFINIÇÕES.

```
concat :: [[α]] -> [α]
concat [ ]      = [ ]
concat (xs:xss) = xs ++ concat xss
```

✓

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

✓

DEMONSTRAÇÃO DE $(\forall xss :: [[Nat]])[\ sum \cdot map\ sum\ xss = sum \cdot concat\ xss\ ]$

$(\forall xss :: [[Nat]])[\ sum\ (map\ sum\ xss) = sum\ (concat\ xss)\ ]$ [(.), 1]

Indução sobre xss. Considere xss = [ ]. Calculemos:

- sum (map sum [ ]) = sum [ ] [map. 1]
  = 0 [sum. 1]
- sum (concat [ ]) = sum [ ] [concat. 1]
  = 0 [sum, 1]

bizarro

já poder parar aqui!

Seja xss :: [[Nat]] t.q. sum (map sum xss) = sum (concat xss) (HI)

Seja xs :: [Nat]. Calculemos:

sum (concat (xs : xss)) = sum (xs ++ concat xss) [concat. 2] ✓
= sum xs + sum (concat xss) [Lema 5] ✓
= sum xs + sum (map sum xss) [HI] ✓
= sum (sum xs : map sum xss) [sum. 2]
= sum (map sum (xs : xss)) [map. 2] ✓

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

4

```
sum :: [Nat] -> Nat
sum []     = 0
sum (x:xs) = x + sum xs
```

(28) **(ii)** DEMONSTRAÇÃO DA (S).

28!

(∀xs, ys :: Nat)[sum (xs ++ ys) = sum xs + sum ys]

Indução sobre xs.

Considere xs = []. Calculemos:

sum ([] ++ ys) = sum ys [(++).1]
= 0 + sum ys [(+)-id]
= sum [] + sum ys [sum.1] ✓

seja xs :: [Nat] tal que sum(xs ++ ys) = sum xs + sum ys. (HI)

seja x :: Nat. Calculemos,

sum ((x:xs) ++ ys) = sum (x : (xs ++ ys)) [(++).2] ✓
= x + sum (xs ++ ys) [sum.2] ✓
= x + (sum xs + sum ys) [HI] ✓
= (x + sum xs) + sum ys [(+)-ass] ✓
= sum (x:xs) + sum ys [sum.2]. ✓

∎

(48) D

16!

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

2 `sum . map sum` = sum . concat
2 `filter p . map f` = map f . filter (p . f)
2♡ `map f . map g` = map (f . g)
2 `map f . take n` = take n . map f
2 `filter p . concat` = concat . map (filter p)
2 `take m . take n` = take (min m n)
2 `concat . map reverse` = ~~reverse . concat~~ concat . concat
2 `sum . map double` = double . sum

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

32

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

DEMONSTRAÇÃO DE ♡ ♡ .

Por indução.

BASE: Calculemos:

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

PASSO INDUTIVO: Seja $x_s : [\alpha]$ tal que (map f . map g) x_s = map (f.g) x_s [H].
Seja $x : \alpha$, calculemos:

(map f . map g) (x : x_s) = map f (map g (x : x_s)) [(.).1]
= map f (g x : map g x_s) [map.2]
= f (g x) : map f (map g x_s) [map.2]
= f (g x) : (map f . map g) x_s [(.).1$^\in$]
= f (g x) : map (f . g) x_s [H]
= (f . g) x : map (f . g) x_s [(.).1$^\in$]
= map (f . g) (x : x_s). [map.2$^\in$]

pulou a (.).1 (2 vezes)

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

α ö

(52) **F2.** *Aja numa maneira melhor:* **(i)** defina a *sum* como caso especial dum conceito mais abstrato capaz de definir todas as funções que o programador considerou nos exemplos acima; **(ii)** enuncie claramente o teorema que precisas demonstrar para ganhar todos esses teoremas de tal programador, de graça;[3] **(iii)** demonstre o teorema que enunciou no **(ii)**.

(6) DEFINIÇÃO LEGAL DA *sum*.

6

```
foldr :: (α → β → β) → β → [α] → β        sum = foldr (+) 0
foldr _ leve [] = leve
foldr op leve (x:xs) = x `op` foldr op leve xs
```

(8) TEOREMA.

8

Sejam α um tipo, op : α → α → α e leve : α. Se op é associativa e leve é op-identidade esquerda, então para todos x_s, y_s : [α],

foldr op leve (x_s ++ y_s) = foldr op leve x_s `op` foldr op leve y_s

(38) DEMONSTRAÇÃO.

38!

Suponha que op é associativa e leve é op-identidade esquerda e seja y_s : [α]. Por indução em x_s.

BASE: Calculemos:

foldr op leve ([] ++ y_s) = foldr op leve y_s [(++).1]
= leve `op` foldr op leve y_s [leve op-id esq]
= foldr op leve [] `op` foldr op leve y_s [foldr.1^E]

PASSO INDUTIVO: Seja x_s : [α] tal que foldr op leve (x_s ++ y_s) = foldr op leve x_s `op` foldr op leve y_s (HI). ~~Calculemos~~: Seja x: α, calculemos:

foldr op leve (x:x_s ++ y_s) = foldr op leve (x : (x_s ++ y_s)) [(++).2]
= x `op` foldr op leve (x_s ++ y_s) [foldr.2]
= x `op` (foldr op leve x_s `op` foldr op leve y_s) [HI]
= (x `op` foldr op leve x_s) `op` foldr op leve y_s [op-assoc]
= foldr op leve (x:x_s) `op` foldr op leve y_s [foldr.2^E]

Só isso mesmo.

[3] Não esqueça incluir as hipoteses necessárias!

(48) **D**

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
sum . map sum          =
filter p . map f       =
map f . map g          =
map f . take n         = take n . map f
filter p . concat      =
take m . take n        = take (n+m)
concat . map reverse   = concat . map
sum . map double       =
```

2

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

DEMONSTRAÇÃO DE _____ .

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

(52) **F2.** *Aja numa maneira melhor:* **(i)** defina a *sum* como caso especial dum conceito mais abstrato capaz de definir todas as funções que o programador considerou nos exemplos acima; **(ii)** enuncie claramente o teorema que precisas demonstrar para ganhar todos esses teoremas de tal programador, de graça;[3] **(iii)** demonstre o teorema que enunciou no **(ii)**.

(6) DEFINIÇÃO LEGAL DA *sum*.

6

```
sum :: [a] → a
sum = foldr (+) 0
```

(8) TEOREMA.

0

```
foldr :: (a → b → b) → b → [a] → b
foldr p v []     = v
foldr p v (x:xs) = p x (foldr p v xs)
```

(38) DEMONSTRAÇÃO.

0

Seja xs' : list a

[+xs'@(x:xs)] sum xs' = foldr (+) 0 xs'

Calculamos:

sum xs' = foldr (+) 0 xs' [foldr.2]

sum xs' = x + (foldr (+) 0 xs) [sum.2]

sum xs' = x + (sum xs).

```
sum :: [a] → a
sum []     = 0
sum (x:xs) = x + (sum xs)
```

Só isso mesmo.

[3] Não esqueça incluir as hipoteses necessárias!

(48) **D**

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
sum . map sum          = SUM . CONCAT
filter p . map f       = FILTER (P.F)
map f . map g          = MAP (F.G)
map f . take n         = TAKE N . MAP F
filter p . concat      = CONCAT . MAP (FILTER P)
take m . take n        = TAKE (MIN N M)
concat . map reverse   = CONCAT . CONCAT
sum . map double       = DOUBLE . SUM
```

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

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

DEMONSTRAÇÃO DE _____ .

Demonstramos `map f . map g = map (f . g)`

$\forall ys, ys :: [c], \forall f, f :: (a \to b), \forall g, g :: (c \to a), \forall xs, x :: [a], \forall zs, z :: [b]$.

Calculemos: `(map f . map g) ys = map (f . g) ys` [H.I.] ?

```
map f . g (y : ys)  = f . g y : map (f . g) ys
map f . g x : map g ys  =
map f xs  =
```

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

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
sum . map sum          = sum ( map sum)
filter p . map f       =
map f . map g          = map (f . g)
map f . take n         = take n (map f)
filter p . concat      =
take m . take n        = take m
concat . map reverse   = concat
sum . map double       = sum (map double)
```

2

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

DEMONSTRAÇÃO DE take m . take n.

2

(take m . take n) x = take m (take n x)
= take m x ?

O valor de n não vai interferir caso m < n; é o quê?!

Caso m > n temos que take m . take n = take n

(por quê?! E como podemos juntar esses dois casos?)

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+\ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+\ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** Definição recursiva da *sum*.

4

```
sum :: List α → α        Nat  Nat
sum []     = 0
sum (x:xs) = x + sum xs
```

(28) **(ii)** Demonstração da (S).

0

```
sum (xs ++ ys) = sum (x:(xs ++ ys))
               = sum ((x:xs) ++ ys)
?              = sum (sum xs ++ ys)
               = sum xs + sum ys
```

?!?

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
sum . map sum          = SUM . REVERSE . MAP SUM . REVERSE
filter p . map f       = REVERSE . FILTER P . REVERSE . MAP F
2 map f . map g        = MAP (F . G)
2 map f . take n       = TAKE N . MAP F
filter p . concat      =
take m . take n        = TAKE (M+1) . TAKE (N-1)
concat . map reverse   =
2 sum . map double     = DOUBLE . SUM
```

CAPS LOCK OFF!

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

DEMONSTRAÇÃO DE _____ .

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+\ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+\ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** Definição recursiva da *sum*.

4

```
SUM :: [NAT] → NAT
SUM [] = 0
SUM (x:xs) = x + (SUM xs)
```

(28) **(ii)** Demonstração da (S).

6

POR INDUÇÃO EM xs:

CASO BASE: SUM([]++ys) ≟ SUM [] + SUM ys ← ALVO

⇒ SUM ys = SUM [] + SUM ys [(++).1] ← tá inferindo algo a partir do teu alvo?!

⇒ SUM ys = 0 + SUM ys [SUM.1]

⇒ SUM ys = SUM ys + 0 [ASSOCIATIVIDADE DA SOMA]

⇒ SUM ys = SUM ys [(+).1] ← parece convincente concluir algo trivial?

PASSO INDUTIVO: ∀ B [SUM(B ++ ys) = SUM B + SUM ys] ⇒ SUM(A:B ++ ys) = SUM A:B + SUM ys

nomes ruins!

SUM(A:(B++ys)) = SUM A:B + SUM ys [(++).2] ← é isso que (++).2 tá dizendo mesmo?

A + SUM(B++ys) = SUM A:B + SUM ys [SUM.2]

A + SUM(B++ys) = A + SUM B + SUM ys [SUM.2]

~~SUM(B++ys) = SUM~~ SUM(B++ys) + A = (SUM B + SUM ys) + A [ASSOCIATIVIDADE DA SOMA]

SUBTRAINDO A DE AMBOS OS LADOS TEM-SE A SUPOSIÇÃO DO PASSO INDUTIVO. LOGO, ESTÁ DEMONSTRADO QUE (S) É VERDADEIRA.

redundante

Type error.
Não usamos o '⇒' assim!

NUNCA usamos aspas assim!!

temos subtração nos Nats?(!)

o que é isso?!
e... não é algo que tens nos dados já?!

sujiro fortemente ver os inicios de cada unidade da minha FMC1 de 2022.2.

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

```
1 sum . map sum          =
2 filter p . map f       =
3 map f . map g          =
4 map f . take n         =
5 filter p . concat      =
6 take m . take n        =
7 concat . map reverse   =
8 sum . map double       =
```

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

DEMONSTRAÇÃO DE _____ .

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

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

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

4

```
sum :: [Nat] → Nat
sum [] = 0
sum (x:xs) = x + sum xs
```

(28) **(ii)** DEMONSTRAÇÃO DA (S).

24

xs ys nunca foram declarados.

sum (xs ++ ys) ? por indução!

não ajuda escrever assim

(*) caso xs = [], calculamos:

= sum ([] ++ ys) [++.1]

= sum ys [++.1]

= (sum ys) + 0 [+.1]

= sum ys + sum [] [sum.1]

= ~~sum ys + sum xs [*]~~

= sum xs + sum ys [comutativo] → *

[]

suponha sum (xs ++ ys) = sum xs + sum ys.

Daí

sum ((x:xs) ++ ys) = sum (x:(xs ++ ys)) [++.2]

= x + sum (xs ++ ys) [+.2]

= x + (sum xs + sum ys) [*] H.I.

= (x + sum xs) + sum ys [Assoc.]

= sum (x:xs) + sum ys [sum.2]

[[1,2,3], [4,5,6]]

[[3,2,1], [6,5,4]]

[3,2,1,6,5,4]

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

2 `sum . map sum` = sum . concat

2 `filter p . map f` = map f . filter (f . p)

2 `map f . map g` = map (f . g)

2 `map f . take n` = take n . map f

2 `filter p . concat` = concat . map (filter p)

2 `take m . take n` = take (min n m)

1 `concat . map reverse` = foldl ((++) . reverse) []

2 `sum . map double` = double . sum

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

```
sum :: [Nat] → Nat
sum [] = 0
sum (x:xs) = x + sum xs
```

```
double → Nat → Nat
double 0 = 0
double (S n) = S (S (double n))
```

```
map :: (Nat → Nat) → [Nat] → [Nat]
map _ [] = []
map f (x:xs) = f x : map f xs
```

22

DEMONSTRAÇÃO DE (∀ l : ListNat) [(sum . map double) l = (double . sum) l]

Indução.

Base.

Calculamos:

(sum . map double) []
= [f comp]
sum (map double [])
= [map.1]
sum []
= [sum.1]
0
= [← double.1]
double 0
= [← sum.1]
double (sum [])
= [← f comp]
(double . sum) []

Passo indutivo.

Seja xs : LN (sum . map double) xs = (double . sum) xs

Seja x : Nat.

Calculamos:

(sum . map double) (x:xs)
= [f comp]
sum (map double (x:xs))
= [map.2] & [double? sem saber a forma de x?
sum ((S (S x)) : map double xs)
= [sum.2]
(S (S x)) + sum (map double xs)
= [← f comp]
(S (S x)) + ((sum . map double) xs)
= [HP]
(S (S x)) + ((double . sum) xs)
= [f comp]
(S (S x)) + double (sum xs)
= [(+) comm]
double (sum xs) + (S (S x))
... :(

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+\ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+\ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

4

```
sum :: [Nat] → Nat
sum [] = 0
sum (x:xs) = x + sum xs
```

(28) **(ii)** DEMONSTRAÇÃO DA (S). (∀ xs, ys : LN) [sum (xs ++ ys) = sum xs + sum ys]

28

Indução.

Base.

Calculemos

sum ([] ++ ys)
= [(++).1]
sum ys
= [(+).1 ∘ (+) sum]
0 + sum ys
= [sum.1]
sum [] + sum ys

Passo indutivo.

Seja xs : LN t.q. sum (xs ++ ys) = sum xs + sum ys, HI

Seja x : Nat.

Calculemos:

sum (x:xs) + sum ys
= [(+) assoc ∘ sum.2]
x + (sum xs + sum ys)
= [hip. ind.]
x + (sum (xs ++ ys))
= [← sum.2]

sum (x : (xs ++ ys))
= [← (++).2]
sum ((x:xs) ++ ys)

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

2 `sum . map sum` = sum . concat

`filter p . map f` = map f . ~~map~~ filter p

2 `map f . map g` = map (f . g)

2 `map f . take n` = take n . map f

`filter p . concat` = filter p . map (++)

• `take m . take n` = take (n+m)

`concat . map reverse` = ~~concat~~ map (++) . map concat

`sum . map double` = double . map sum.

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

DEMONSTRAÇÃO DE ~~X~~ _____ .

8

Façamos indução em n e m.

Base:

(take 0 . take 0) xs = take 0 (take 0 xs) [(.).1]

= take 0 [] [take.1]

= take (0+0) [] [(+).1←]

necessárias!

Passo indutivo:

Sejam u, v ∈ Nat, tal que take u . take v = take (u+v).

Sejam x : a e xs : [a]. Calculemos

take su . take sv (x:xs) = take su (take sv (x:xs)) [(.).1]

= take su x:(take v xs) [take.2]

= x: take u (take v xs) [take.3]

= x: (take u . take v) xs [(.).1←]

= x: (take (u+v)) xs [H.I]

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

(52) **F**

Escolha **até 1** dos **F1** e **F2**.

Um programador definiu a *sum* recursivamente e demonstrou a propriedade seguinte:

$$sum\ (xs +\!\!+\ ys) = sum\ xs + sum\ ys. \tag{S}$$

Depois definiu a *product*, também recursivamente, e demonstrou a propriedade

$$product\ (xs +\!\!+\ ys) = product\ xs \cdot product\ ys.$$

E depois a mesma coisa sobre a *concat*, etc., etc.

(32) **F1.** Faça a primeira coisa que ele fez:

(4) **(i)** DEFINIÇÃO RECURSIVA DA *sum*.

```
Sum :: [Nat] -> Nat
sum [] = 0
Sum (x:xs) = x + Sum xs
```

4

(28) **(ii)** DEMONSTRAÇÃO DA (S).

24

Façamos indução em xs.

Base

sum([]++ys) = Sum ys	[(++).1]
= Sum ys + 0	[(+).1$^{\leftarrow}$]
= Sum ys + Sum []	[Sum.1$^{\leftarrow}$]
= Sum [] + Sum ys	[(+)-com]

Passo indutivo:

Sejam xs, ys : [α] tal que sum(xs ++ ys) = Sum xs + Sum ys ✓

Seja x : α. ✓

Calculemos:

Sum ((x:xs)++ys) = Sum (x:(xs++ys))	[(++).2] ✓
= x + Sum (xs++ys)	[Sum.2] ✓
= x + (Sum xs + Sum ys)	[H.I] ✓
⋮	
= sum (x:xs) + sum ys	

ainda não!

succ double
[[1,2],[

~~[[1,2],[3,5]]~~

~~[1,2,3,4] → [2,4,6,8]~~

["oi", "mundo", "bom"]

io odnum mob

f: Int → Int
p: Int → Bool
p.f: Int → Bool

["ab", "ef"]
"bafe"

(48) D

(16) **D1.** Complete as igualdades seguintes **com algo interessante:**[2]

2 `sum . map sum` = sum . concat
`filter p . map f` = filter (p.f)
2 `map f . map g` = map (f.g)
2 `map f . take n` = take n . map f
2 `filter p . concat` = concat . map (filter p)
2 `take m . take n` = ~~take (m+n)~~ take (min m n)
2 `concat . map reverse` = concat . concat
2 `sum . map double` = double . sum
concat

(32) **D2.** Escolha **exatamente uma** delas para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto aquelas que são definidas na primeira página.

DEFINIÇÕES.

32:

use a convenção map.1 etc. para não precisar rotular!

~~Todo mapfg~~

```
map :: (a → b) → [a] → [b]
map _ [] = []                    (map 1)
map f (x:xs) = fx : map f xs     (map 2)
```

DEMONSTRAÇÃO DE `map f . map g = map (f . g)`

Sejam $f: b \to c$, $g: a \to b$. Provarei por indução em listas que $(map\ f . map\ g)\ xs' = (map\ (f.g))\ xs'$ para todo $xs' \in [a]$.

Base ($xs' = []$):

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

Indução: suponha que $xs' = x:xs$ e que $(map\ f . map\ g)\ xs = map\ (f.g)\ xs$ (HI)

```
(map f . map g) (x:xs) = map f (map g (x:xs))   [def .]
                       = map f (g x : map g xs) [map 2]
                       = f (g x) : map f (map g xs)  [map 2]
                       = f (g x) : map (f.g) xs      [def .] + (HI)
                       = (f.g) x : map (f.g) xs      [def .]
                       = map (f.g) (x:xs)
```

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

(52) **F2.** *Aja numa maneira melhor:* **(i)** defina a *sum* como caso especial dum conceito mais abstrato capaz de definir todas as funções que o programador considerou nos exemplos acima; **(ii)** enuncie claramente o teorema que precisas demonstrar para ganhar todos esses teoremas de tal programador, de graça;[3] **(iii)** demonstre o teorema que enunciou no **(ii)**.

(6) DEFINIÇÃO LEGAL DA *sum*.

6

Considerando as hipóteses do teorema, temos sum = op, + = *, 0 = ε e Nat = m.

→ qual a vantagem dela?

Not really! ÷

(8) TEOREMA.

→ veja a res. na p31 deste PDF.

→ Bad name, I only realized it after writing the whole thing

6

Seja m monoide com operação *: m → m → m e elemento neutro ε. Seja op: [m] → m definida por op [] = ε e op (x:xs) = x * op xs. Então op (xs ++ ys) = op xs * op ys para todos xs, ys ∈ [m].

(38) DEMONSTRAÇÃO.

38

Prova por indução em ~~listas~~ xs.

Base (xs = []): op (xs ++ ys) = op ([] ++ ys)
= op ys
= ε * op ys
= op [] * op ys
= op xs * op ys

def xs
def ++
m monoide
def op
def xs

melhor começar aqui

Passo indutivo: suponha xs = x:xs', op (xs' ++ ys) = op xs' * op ys. (HI)

op (xs ++ ys) = op (x:xs' ++ ys)
= op (x:(xs' ++ ys))
= x * op (xs' ++ ys)
= x * (op xs' * op ys)
= (x * op xs') * op ys
= op (x:xs') * op ys
= op xs * op ys

def xs
def ++
def op
HI
m monoide
def op
def xs

e terminar aqui

Só isso mesmo.

[3] Não esqueça incluir as hipoteses necessárias!