

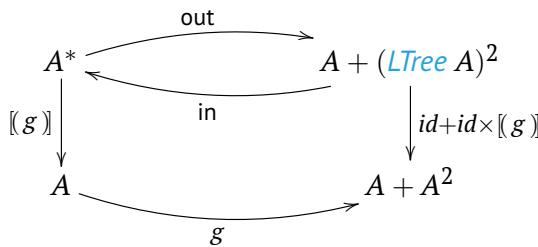
# Cálculo de Programas

## Resolução - Ficha 10

Eduardo Freitas Fernandes

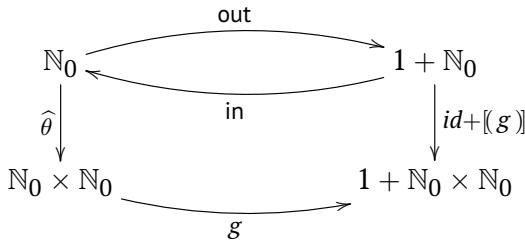
2026

### Exercício 1



O anamorfismo de  $g$  inverte uma lista, ou seja, é a função *reverse*.

### Exercício 2



$$\begin{aligned} g &:: (\text{Int}, \text{Int}) \rightarrow () + (\text{Int}, \text{Int}) \\ g &= \widehat{(\leqslant)} \rightarrow i_1 \cdot (!), i_2 \cdot (id \times \text{succ}) \end{aligned}$$

### Exercício 3

$$\begin{aligned} length \cdot concat &= sum \cdot map \ length \\ &\equiv \{ \text{def. concat, def. sum} \} \\ length \cdot \langle [nil, conc] \rangle &= \langle [0, add] \rangle \cdot map \ length \\ &\equiv \{ \text{absorção-cata} \} \\ length \cdot \langle [nil, conc] \rangle &= \langle [0, add] \cdot (id + length \times id) \rangle \\ &\Leftarrow \{ \text{absorção-+, fusão-cata} \} \\ length \cdot [nil, conc] &= [0, add \cdot (length \times id)] \cdot (id + id \times length) \\ &\equiv \{ \text{fusão-+, absorção-+, eq-+} \} \\ &\left\{ \begin{array}{l} length \cdot nil = 0 \\ length \cdot conc = add \cdot (length \times id) \cdot (id \times length) \end{array} \right. \end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{functor-}\times \} \\
&\quad length \cdot conc = add \cdot (length \times length) \\
&\equiv \{ \text{??????} \} \\
&\quad true \\
&\square
\end{aligned}$$

### Exercício 4

$$\begin{aligned}
length &= sum \cdot (\text{map } \underline{1}) \\
&\equiv \{ \text{def. } sum \} \\
length &= (\underline{[0, add]} \cdot \text{map } \underline{1}) \\
&\equiv \{ \text{absorção-cata} \} \\
length &= (\underline{[0, add]} \cdot B(\underline{1}, id)) \\
&\equiv \{ \text{def. bi-functor de listas} \} \\
length &= (\underline{[0, add]} \cdot (id + \underline{1} \times id)) \\
&\equiv \{ \text{absorção-+} \} \\
length &= (\underline{[0, add \cdot (\underline{1} \times id)]}) \\
&\equiv \{ add \cdot (\underline{1} \times id) = succ \cdot \pi_2 \} \\
length &= (\underline{[0, succ \cdot \pi_2]}) 
\end{aligned}$$

Podemos verificar que  $add \cdot (\underline{1} \times id)$  é equivalente a  $succ \cdot \pi_2$ , pois esta recebe um par de valores, destroi o primeiro (transformando no valor 1) e mantém o segundo, de seguida soma o segundo a 1, obtendo assim o valor equivalente ao seu sucessor.

$$\begin{aligned}
length &= length \cdot (\text{map } f) \\
&\equiv \{ \text{def. } length \} \\
length &= (\underline{[0, succ \cdot \pi_2]} \cdot (\text{map } f)) \\
&\equiv \{ \text{absorção-cata} \} \\
length &= (\underline{[0, succ \cdot \pi_2]} \cdot B(f, id)) \\
&\equiv \{ \text{def. bi-functor de listas} \} \\
length &= (\underline{[0, succ \cdot \pi_2]} \cdot (id + f \times id)) \\
&\equiv \{ \text{absorção-+, natural-id} \} \\
length &= (\underline{[0, succ \cdot \pi_2 \cdot (f \times id)]}) \\
&\equiv \{ \text{natural-}\pi_2 \} \\
length &= (\underline{[0, succ \cdot id \cdot \pi_2]}) \\
&\equiv \{ \text{natural-id} \} \\
length &= (\underline{[0, succ \cdot \pi_2]}) 
\end{aligned}$$

### Exercício 5

$$\begin{aligned}
depth \cdot LTree f &= depth \\
&\equiv \{ \text{def. } depth \}
\end{aligned}$$

$$\begin{aligned}
& (\lfloor \underline{1}, \text{succ} \cdot \text{umax} \rfloor) \cdot T f = \text{depth} \\
\equiv & \quad \{ \text{absorção-cata} \} \\
& (\lfloor \underline{1}, \text{succ} \cdot \text{umax} \rfloor \cdot B(f, id)) = \text{depth} \\
\equiv & \quad \{ \text{def. bi-functor de LTree} \} \\
& (\lfloor \underline{1}, \text{succ} \cdot \text{umax} \rfloor \cdot (f + (id \times id))) = \text{depth} \\
\equiv & \quad \{ \text{functor-id-}\times, \text{absorção-}+ \} \\
& (\lfloor \underline{1} \cdot f, \text{succ} \cdot \text{umax} \cdot (id \times id) \rfloor) = \text{depth} \\
\equiv & \quad \{ \text{fusão-const, natural-}id \} \\
& (\lfloor \underline{1}, \text{succ} \cdot \text{umax} \rfloor) = \text{depth}
\end{aligned}$$

## Exercício 6

$$\begin{aligned}
& \text{bubble } (x : y : xs) \\
| & x > y = y : \text{bubble } (x : xs) \\
| & \text{otherwise} = x : \text{bubble } (y : xs) \\
\text{bubble } x & = x
\end{aligned}$$

O primeiro passo será substituir o nome da função por *divide* e de seguida remover as chamadas recursivas:

$$\begin{aligned}
& \text{divide } (x : y : xs) \\
| & x > y = y \dots (x : xs) \\
| & \text{otherwise} = x \dots (y : xs) \\
\text{divide } x & = x
\end{aligned}$$

De seguida emparelhamos o resultado e por fim injetamos o resultado num co-produto, dado que existem dois tipos de resultado:

$$\begin{aligned}
& \text{divide } (x : y : xs) \\
| & x > y = i_2(y, (x : xs)) \\
| & \text{otherwise} = i_2(x, (y : xs)) \\
\text{divide } x & = i_1 x
\end{aligned}$$

Podemos então inferir o tipo da função *divide*:

$$\text{divide} : A^* \rightarrow A^* + A \times A^*$$

Verificamos que o bi-functor necessário para formar este hilomorfismo será o das **Listas com Sentinelas**:

$$\text{data } SList\ a\ b = Stl\ b \mid Cons\ a\ (SList\ a\ b)$$

Temos então:

$$B(Z, X, Y) = Z + X + Y$$

Pelo tipo de *divide* podemos inferir o tipo de *conquer*:

$$\begin{aligned}
& \text{conquer} :: [a] + (a, [a]) \rightarrow [a] \\
& \text{conquer} = [id, cons]
\end{aligned}$$

## Exercício 7

```
data Point a = Point {x :: a, y :: a, z :: a} deriving (Eq, Show)
outPoint = ⟨⟨x, y⟩, z⟩
-- inPoint = uncurry (uncurry Point)
inPoint ((a, b), c) = Point a b c
```

## Exercício 8

```
module B_Tree where
import Cp
-- (1) Datatype definition
data B_Tree a = Nil | Block {leftmost :: B_Tree a, block :: [(a, B_Tree a)]}
deriving (Show)
inB_Tree :: () + (B_Tree a, [(a, B_Tree a)]) → B_Tree a
inB_Tree = [Nil, Block]
outB_Tree :: B_Tree a → () + (B_Tree a, [(a, B_Tree a)])
outB_Tree Nil = i1 ()
outB_Tree (Block l b) = i2 (l, b)
baseB_Tree g f = id + (f × (map (g × f)))
-- (2) Ana + cata + hylo
recB_Tree f = baseB_Tree id f
(|g|) = g · (recB_Tree (|g|)) · outB_Tree
anaB_Tree g = inB_Tree · (recB_Tree (anaB_Tree g)) · g
hyloB_Tree f g = (|f|) · anaB_Tree g
-- (3) Map
instance Functor B_Tree where
fmap f = (|inB_Tree · baseB_Tree f id|)
-- (4) Examples
-- (4.1) Count and depth
countB_Tree = (|[zero,
add · (id × sum · map (succ · π2))]|)
depthB_Tree = (|[zero,
succ · umax · (id × maximum · map π2)]|)
-- (4.2) Serialization
-- in-order traversal
inordtB_Tree = (|[nil, conc · (id × (concat · map cons))]|)
-- pre-order traversal
preordtB_Tree = (|[nil, aux]|)
where
aux (l, []) = l
aux (l, (h, a) : t) = (h : l) ++ a ++ (concat (map cons t))
-- post-order traversal
postordtB_Tree = (|[nil, conc · (id × (concat · map aux))]|)
where
aux (x, l) = l ++ [x]
```