

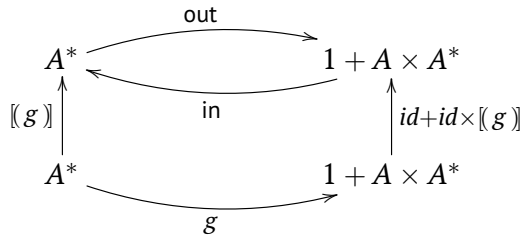
Cálculo de Programas

Resolução - Ficha 10

Eduardo Freitas Fernandes

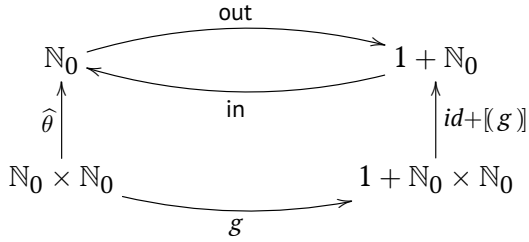
2026

Exercício 1



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

Exercício 2



$$g :: (Int, Int) \rightarrow () + (Int, Int)$$

$$g = (\widehat{\leq}) \rightarrow i_1 \cdot (!), i_2 \cdot (id \times succ)$$

Exercício 3

$$\begin{aligned}
 & length \cdot concat = sum \cdot map \ length \\
 \equiv & \quad \{ \text{def. } concat, \text{ def. } sum \} \\
 & length \cdot \llbracket [nil, conc] \rrbracket = \llbracket [0, add] \rrbracket \cdot map \ length \\
 \equiv & \quad \{ \text{absorção-cata} \} \\
 & length \cdot \llbracket [nil, conc] \rrbracket = \llbracket [0, add] \rrbracket \cdot (id + length \times id) \\
 \Leftarrow & \quad \{ \text{absorção-+}, \text{ fusão-cata} \} \\
 & length \cdot [nil, conc] = [0, add \cdot (length \times id)] \cdot (id + id \times length) \\
 \equiv & \quad \{ \text{fusão-+}, \text{ absorção-+}, \text{ eq-+} \} \\
 & \begin{cases} length \cdot nil = 0 \\ length \cdot conc = add \cdot (length \times id) \cdot (id \times length) \end{cases}
 \end{aligned}$$

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

Exercício 4

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

Podemos verificar que $\text{add} \cdot (\underline{1} \times \text{id})$ é equivalente a $\text{succ} \cdot \pi_2$, 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.

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

Exercício 5

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

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

Exercício 6

```

bubble (x : y : xs)
  | x > y = y : bubble (x : xs)
  | otherwise = x : bubble (y : xs)
bubble x = x

```

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

```

divide (x : y : xs)
  | x > y = y ... (x : xs)
  | otherwise = x ... (y : xs)
divide x = x

```

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

```

divide (x : y : xs)
  | x > y = i2 (y, (x : xs))
  | otherwise = i2 (x, (y : xs))
divide x = i1 x

```

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 Sentinela**:

```

data SList a b = Stl b | Cons a (SList a b)

```

Temos então:

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

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

```

conquer :: [a] + (a, [a]) -> [a]
conquer = [id, cons]

```

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,  $\widehat{\text{Block}}$ ]
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]
```