

Cálculo de Programas

Algebra of Programming

UNIVERSIDADE DO MINHO
Lic. em Engenharia Informática (3º ano)
Lic. Ciências da Computação (2º ano)

2022/23 - Ficha (*Exercise sheet*) nr. 9

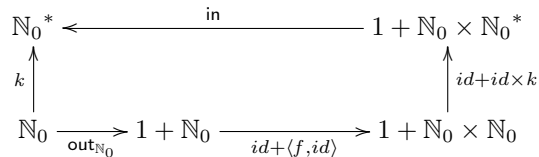
1. Um *anamorfismo* é um “*catamorfismo ao contrário*”, i.e. uma função $k : A \rightarrow T$ tal que *An anamorphism is a “reverse catamorphism”, i.e. a function $k : A \rightarrow T$ such that*

$$k = \text{in} \cdot F k \cdot g \tag{F1}$$

escrevendo-se $k = \llbracket g \rrbracket$. Mostre que o anamorfismo de listas *One writes $k = \llbracket g \rrbracket$. Show that the list-anamorphism*

$$k = \llbracket (\text{id} + \langle f, \text{id} \rangle) \cdot \text{out}_{\mathbb{N}_0} \rrbracket \tag{F2}$$

descrito pelo diagrama *depicted in diagram*



é a função *is the function*

$$\begin{aligned}
 k \ 0 &= [] \\
 k \ (n + 1) &= (2 \ n + 1) : k \ n
 \end{aligned}$$

para $f \ n = 2 \ n + 1$. (Que faz esta função?) *for $f \ n = 2 \ n + 1$. (What does this function do?)*

2. A função *concat*, extraída do *Prelude* do Haskell, é o catamorfismo de listas *The concat function, taken from the Haskell Prelude, is the list-catamorphism*

$$\text{concat} = \llbracket [\text{nil}, \text{conc}] \rrbracket \tag{F3}$$

onde $\text{conc} \ (x, y) = x \ ++ \ y$ e $\text{nil} \ _ = []$. Apresente justificações para a prova da propriedade *where $\text{conc} \ (x, y) = x \ ++ \ y$ and $\text{nil} \ _ = []$. Provide justifications for proof of property*

$$\text{length} \cdot \text{concat} = \text{sum} \cdot \text{map length} \tag{F4}$$

que a seguir se apresenta, onde é de esperar que as leis de fusão-cata e absorção-cata desempenhem um papel importante:

which is presented below, where the cata-fusion and cata-absorption laws are expected to play an important role:

$$\begin{aligned}
 & \text{length} \cdot \text{concat} = \text{sum} \cdot \text{map length} \\
 \equiv & \{ \dots \} \\
 & \text{length} \cdot \llbracket [\text{nil}, \text{conc}] \rrbracket = \llbracket [\underline{0}, \text{add}] \rrbracket \cdot \text{map length} \\
 \equiv & \{ \dots \} \\
 & \text{length} \cdot \llbracket [\text{nil}, \text{conc}] \rrbracket = \llbracket [\underline{0}, \text{add}] \rrbracket \cdot (\text{id} + \text{length} \times \text{id}) \\
 \Leftarrow & \{ \dots \} \\
 & \text{length} \cdot [\text{nil}, \text{conc}] = [\underline{0}, \text{add} \cdot (\text{length} \times \text{id})] \cdot (\text{id} + \text{id} \times \text{length}) \\
 \equiv & \{ \dots \} \\
 & \begin{cases} \text{length} \cdot \text{nil} = \underline{0} \\ \text{length} \cdot \text{conc} = \text{add} \cdot (\text{length} \times \text{id}) \cdot (\text{id} \times \text{length}) \end{cases} \\
 \equiv & \{ \dots \} \\
 & \text{length} \cdot \text{conc} = \text{add} \cdot (\text{length} \times \text{length}) \\
 \equiv & \{ \dots \} \\
 & \text{true} \\
 & \square
 \end{aligned}$$

3. Recorra à lei da absorção-cata, entre outras, para verificar as seguintes propriedades sobre listas

Use the cata-absorption law, among others, to prove the following properties about lists

$$\text{length} = \text{sum} \cdot (\text{map } \underline{1}) \tag{F5}$$

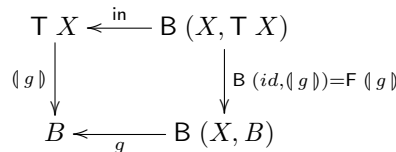
$$\text{length} = \text{length} \cdot (\text{map } f) \tag{F6}$$

onde length, sum e map são catamorfismos de listas que conhece. (Recorda-se que o bifunctor de base para listas é $B(f, g) = \text{id} + f \times g$, de onde se deriva $F f = B(\text{id}, f) = \text{id} + \text{id} \times f$.)

where length, sum and map they are list-catamorphisms you know. (Remember that the basic bifunctor for lists is $B(f, g) = \text{id} + f \times g$, yielding $F f = B(\text{id}, f) = \text{id} + \text{id} \times f$.)

4. O diagrama genérico de um catamorfismo de gene g sobre o tipo paramétrico $T X \cong B(X, T X)$ cuja base é o bifunctor B , bem como a sua propriedade universal, são representados a seguir:

The generic diagram of a catamorphism with gene g over the parametric type $T X \cong B(X, T X)$ with base B , as well as its universal property, are represented below:



$$k = \llbracket g \rrbracket \equiv k \cdot \text{in} = g \cdot \underbrace{B(\text{id}, k)}_{F k}$$

De seguida, apresenta-se uma revisão do inventário de tipos indutivos da questão 6 da ficha anterior, recorrendo agora aos seus functores de base:

Next, a review of the inventory of inductive types of question 6 of the previous exercise sheet is given, now using its base-functors:

(a) Árvores com informação de tipo A nos nós (*Trees whose data of type A are stored in their nodes*):

$$T = \text{BTree } A \quad \begin{cases} \mathbf{B}(X, Y) = 1 + X \times Y^2 \\ \mathbf{B}(g, f) = id + g \times f^2 \end{cases} \quad \text{in} = [\text{Empty}, \text{Node}]$$

Haskell: `data BTree a = Empty | Node (a, (BTree a, BTree a))`

(b) Árvores com informação de tipo A nas folhas (*Trees with data in their leafs*):

$$T = \text{LTree } A \quad \begin{cases} \mathbf{B}(X, Y) = X + Y^2 \\ \mathbf{B}(g, f) = g + f^2 \end{cases} \quad \text{in} = [\text{Leaf}, \text{Fork}]$$

Haskell: `data LTree a = Leaf a | Fork (LTree a, LTree a)`

(c) Árvores com informação nos nós e nas folhas (*Full trees — data in both leaves and nodes*):

$$T = \text{FTree } B \ A \quad \begin{cases} \mathbf{B}(Z, X, Y) = Z + X \times Y^2 \\ \mathbf{B}(h, g, f) = h + g \times f^2 \end{cases} \quad \text{in} = [\text{Unit}, \text{Comp}]$$

Haskell: `data FTree b a = Unit b | Comp (a, (FTree b a, FTree b a))`

(d) Árvores de expressão (*Expression trees*):

$$T = \text{Expr } V \ O \quad \begin{cases} \mathbf{B}(Z, X, Y) = Z + X \times Y^* \\ \mathbf{B}(h, g, f) = h + g \times \text{map } f \end{cases} \quad \text{in} = [\text{Var}, \text{Op}]$$

Haskell: `data Expr v o = Var v | Op (o, [Expr v o])`

Partindo da definição *genérica* de `map` associado ao tipo T ,

Starting from the generic definition of `map` associated with the type T ,

$$T \ f = (\text{in} \cdot \mathbf{B}(f, id))$$

calcule $fmap \ f = T \ f$ para $T := \text{BTree}$, entregando o resultado em Haskell sem combinadores *pointfree*. (Repare-se que se tem sempre $F \ k = \mathbf{B}(id, k)$.)

derive $fmap \ f = T \ f$ for $T := \text{BTree}$, delivering the result in Haskell without point-free combinators. (Note that we always have $F \ k = \mathbf{B}(id, k)$.)

5. Seja dado o catamorfismo

Let catamorphism

$$depth = (\text{[one], succ} \cdot \text{umax})$$

que dá a profundidade de árvores do tipo LTree , onde $umax(a, b) = \max a \ b$. Mostre, por absorção-cata, que a profundidade de uma árvore t não é alterada quando aplica uma função f a todas as suas folhas:

be given, which gives the depth of trees of type LTree , where $umax(a, b) = \max a \ b$. Show, by cata-absorption, that the depth of a tree t is not changed when you apply a function f to all its leaves:

$$depth \cdot \text{LTree } f = depth \tag{F7}$$