DIUM – LESI/LMCC
Métodos de Programação I
Exercícios Resolvidos

Carlos Bacelar, Luís Barbosa,
José Barros, Alcino Cunha, Maria João Frade,
Luís Neves, José Nuno Oliveira, Jorge Sousa Pinto, Nuno Rodrigues

Dezembro 2004

Capítulo 1

Cálculo Não-recursivo

I – Produtos e Coprodutos

1. Use a definição $f \times g \ = \ \langle f \cdot \pi_1, g \cdot \pi_2 \rangle$ para provar que a propriedade

$$(g \cdot h) \times (i \cdot j) \ = \ (g \times i) \cdot (h \times j)$$

se verifica.

Resolução:
Basta trabalhar apenas o lado direito da igualdade:

$$\begin{array}{ll} & (g \cdot h) \times (i \cdot j) = (g \times i) \cdot (h \times j) \\ \Leftrightarrow & \quad \{\text{pela definição dada}\} \\ & (g \cdot h) \times (i \cdot j) = \langle g \cdot \pi_1, i \cdot \pi_2 \rangle \cdot (h \times j) \\ \Leftrightarrow & \quad \{\text{por fusão-}\times\} \\ & (g \cdot h) \times (i \cdot j) = \langle (g \cdot \pi_1) \cdot (h \times j), (i \cdot \pi_2) \cdot (h \times j) \rangle \\ \Leftrightarrow & \quad \{\text{associatividade da composição}\} \\ & (g \cdot h) \times (i \cdot j) = \langle g \cdot (\pi_1 \cdot (h \times j)), i \cdot (\pi_2 \cdot (h \times j)) \rangle \\ \Leftrightarrow & \quad \{\pi_1 \cdot (a \times b) = a \cdot \pi_1 \text{ e } \pi_2 \cdot (a \times b) = b \cdot \pi_2\} \\ & (g \cdot h) \times (i \cdot j) = \langle g \cdot (h \cdot \pi_1), i \cdot (j \cdot \pi_2) \rangle \\ \Leftrightarrow & \quad \{\text{associatividade da composição de novo}\} \\ & (g \cdot h) \times (i \cdot j) = \langle (g \cdot h) \cdot \pi_1, (i \cdot j) \cdot \pi_2 \rangle \\ \Leftrightarrow & \quad \{\text{pela definição dada, da direita para a esquerda}\} \\ & (g \cdot h) \times (i \cdot j) = (g \cdot h) \times (i \cdot j) \\ \Leftrightarrow & \quad \{\text{propriedade reflexiva da igualdade}\} \\ & \text{verdadeiro} \end{array}$$

2. Aplique a *lei da troca*, $[\langle f, g\rangle, \langle h, k\rangle] = \langle [f, h], [g, k]\rangle$, à definição

$$undistr \;=\; [id \times i_1, id \times i_2]$$

Resolução:
Ter-se-á:

$$\begin{array}{ll} & undistr = [id \times i_1, id \times i_2] \\ \Leftrightarrow & \quad \{\text{definição de produto}\} \\ & [\langle id \cdot \pi_1, i_1 \cdot \pi_2\rangle, \langle id \cdot \pi_1, i_2 \cdot \pi_2\rangle] \\ \Leftrightarrow & \quad \{\text{lei da troca}\} \\ & \langle [id \cdot \pi_1, id \cdot \pi_1], [i_1 \cdot \pi_2, i_2 \cdot \pi_2]\rangle \\ \Leftrightarrow & \quad \{\text{ função identidade}\} \\ & \langle [\pi_1, \pi_1], [i_1 \cdot \pi_2, i_2 \cdot \pi_2]\rangle \end{array}$$

7. Considere a seguinte definição de uma função t, em Haskell:

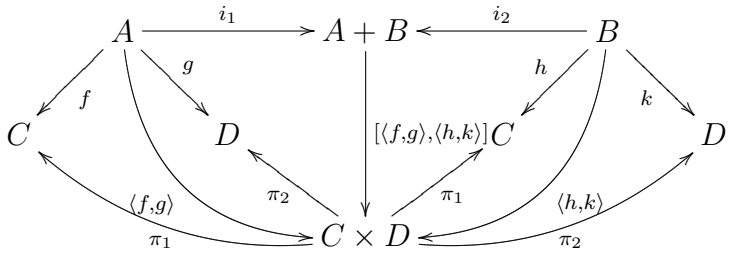
```
t f g h k = [either (split f g)(split h k),
             split (either f h)(either g k)]
```

Qual é o tipo de t? Justifique convenientemente a sua resposta.

Resolução:
Aplicando a lei da troca ($[\langle f, g\rangle, \langle h, k\rangle] = \langle [f, h], [g, k]\rangle$) temos que

```
either (split f g)(split h k) = split (either f h)(either g k)
```

Tendo o seguinte diagrama



logo `t` tem o seguinte tipo

```
t :: (a -> c) -> (a -> d) -> (b -> c) -> (b -> d) -> [Either a b
-> (c, d)]
```

II – Isomorfismos

12. Demonstre a seguinte igualdade

$$[id \times i_1, id \times i_2] = \langle [\pi_1, \pi_1], \pi_2 + \pi_2 \rangle$$

Qual o isomorfismo que esta função estabelece?

Resolução:

$$
\begin{array}{ll}
 & [id \times i_1, id \times i_2] \\
\Leftrightarrow & \quad \{\text{definição de } id\} \\
 & id \cdot [id \times i_1, id \times i_2] \\
\Leftrightarrow & \quad \{\text{reflexão-}\times\} \\
 & \langle \pi_1, \pi_2 \rangle \cdot [id \times i_1, id \times i_2] \\
\Leftrightarrow & \quad \{\text{fusão-}\times\} \\
 & \langle \pi_1 \cdot [id \times i_1, id \times i_2], \pi_2 \cdot [id \times i_1, id \times i_2] \rangle \\
\Leftrightarrow & \quad \{\text{fusão-}+\} \\
 & \langle [\pi_1 \cdot (id \times i_1), \pi_1 \cdot (id \times i_2)], [\pi_2 \cdot (id \times i_1), \pi_2 \cdot (id \times i_2)] \rangle \\
\Leftrightarrow & \quad \{\text{definição de } id\} \\
 & \langle [\pi_1 \cdot ((id \times i_1) \cdot id), \pi_1 \cdot ((id \times i_2) \cdot id)], [\pi_2 \cdot ((id \times i_1) \cdot id), \pi_2 \cdot ((id \times i_2) \cdot id)] \rangle \\
\Leftrightarrow & \quad \{\text{reflexão-}\times\} \\
 & \langle [\pi_1 \cdot ((id \times i_1) \cdot \langle \pi_1, \pi_2 \rangle), \pi_1 \cdot ((id \times i_2) \cdot \langle \pi_1, \pi_2 \rangle)], \\
 & \quad [\pi_2 \cdot ((id \times i_1) \cdot \langle \pi_1, \pi_2 \rangle), \pi_2 \cdot ((id \times i_2) \cdot \langle \pi_1, \pi_2 \rangle)] \rangle \\
\Leftrightarrow & \quad \{\text{absorção-}\times\} \\
 & \langle [\pi_1 \cdot \langle id \cdot \pi_1, i_1 \cdot \pi_2 \rangle, \pi_1 \cdot \langle id \cdot \pi_1, i_2 \cdot \pi_2 \rangle], [\pi_2 \cdot \langle id \cdot \pi_1, i_1 \cdot \pi_2 \rangle, \pi_2 \cdot \langle id \cdot \pi_1, i_2 \cdot \pi_2 \rangle] \rangle \\
\Leftrightarrow & \quad \{\text{cancelamento-}\times, \text{ definição de } id\} \\
 & \langle [\pi_1, \pi_1], [i_1 \cdot \pi_2, i_2 \cdot \pi_2] \rangle \\
\Leftrightarrow & \quad \{\text{absorção-}+\} \\
 & \langle [\pi_1, \pi_1], [i_1, i_2] \cdot (\pi_2 + \pi_2) \rangle \\
\Leftrightarrow & \quad \{\text{reflexão-}+\} \\
 & \langle [\pi_1, \pi_1], id \cdot (\pi_2 + \pi_2) \rangle \\
\Leftrightarrow & \quad \{\text{definição de } id\} \\
 & \langle [\pi_1, \pi_1], \pi_2 + \pi_2 \rangle
\end{array}
$$

A igualdade testemunha o seguinte isomorfismo

$$A \times (B + C) \qquad \cong \qquad (A \times B) + (A \times C)$$

13. Seja *distr* (ler: *distribute right*) a bijecção que estabelece o isomorfismo $A \times (B+C) \cong A \times B + A \times C$.

    (a) Preencha as reticências no diagrama que se segue por forma a ver nele especificada a bijeccão *distl* (ler: *distribute left*) que estabelece o isomorfismo $(B + C) \times A \cong B \times A + C \times A$:

$$(B + C) \times A \xrightarrow{swap} \cdots \xrightarrow{distr} \cdots \xrightarrow{\cdots} B \times A + C \times A$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{distl}$$

    (b) Mostre que

$$[g, h] \times f \quad = \quad [g \times f, h \times f] \cdot distl$$

    é uma propriedade válida sobre *distl*, aplicando, entre outras leis que conhece, as seguintes:

$$\begin{aligned} f \times [g, h] &= [f \times g, f \times h] \cdot distr \\ swap \cdot (f \times g) &= (g \times f) \cdot swap \end{aligned}$$

Resolução:

(a)

$$(B + C) \times A \xrightarrow{swap} A \times (B + C) \xrightarrow{distr} A \times B + A \times C \xrightarrow{swap+swap} B \times A + C \times A$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{distl}$$

(b)

$$\begin{aligned} f \times [g, h] &= [f \times g, f \times h] \cdot distr && (1.1) \\ swap \cdot (f \times g) &= (g \times f) \cdot swap && (1.2) \end{aligned}$$

$$
\begin{array}{ll}
 & [g \times f, h \times f] \cdot distl \\
\Leftrightarrow & \quad \{\text{definição de } distl \text{ inferida na alínea (a)}\} \\
 & [g \times f, h \times f] \cdot ((swap + swap) \cdot (distr \cdot swap)) \\
\Leftrightarrow & \quad \{\text{propriedade associativa da composição de funções}\} \\
 & ([g \times f, h \times f] \cdot (swap + swap)) \cdot (distr \cdot swap) \\
\Leftrightarrow & \quad \{\text{absorção-+}\} \\
 & [(g \times f) \cdot swap, (h \times f) \cdot swap] \cdot (distr \cdot swap) \\
\Leftrightarrow & \quad \{\text{por 1.2}\} \\
 & [swap \cdot (f \times g), swap \cdot (f \times h)] \cdot (distr \cdot swap) \\
\Leftrightarrow & \quad \{\text{fusão-+}\} \\
 & (swap \cdot [f \times g, f \times h]) \cdot (distr \cdot swap) \\
\Leftrightarrow & \quad \{\text{propriedade associativa da composição de funções}\} \\
 & swap \cdot (([f \times g, f \times h] \cdot distr) \cdot swap) \\
\Leftrightarrow & \quad \{\text{por 1.1}\} \\
 & swap \cdot ((f \times [g, h]) \cdot swap) \\
\Leftrightarrow & \quad \{\text{por 1.2}\} \\
 & swap \cdot (swap \cdot ([g, h] \times f)) \\
\Leftrightarrow & \quad \{\text{propriedade associativa da composição de funções}\} \\
 & (swap \cdot swap) \cdot ([g, h] \times f) \\
\Leftrightarrow & \quad \{\text{definição de swap}\} \\
 & (\langle \pi_2, \pi_1 \rangle \cdot \langle \pi_2, \pi_1 \rangle) \cdot ([g, h] \times f) \\
\Leftrightarrow & \quad \{\text{fusão-}\times\} \\
 & \langle \pi_2 \cdot \langle \pi_2, \pi_1 \rangle, \pi_1 \cdot \langle \pi_2, \pi_1 \rangle \rangle \cdot ([g, h] \times f) \\
\Leftrightarrow & \quad \{\text{cancelamento-}\times\} \\
 & \langle \pi_1, \pi_2 \rangle \cdot ([g, h] \times f) \\
\Leftrightarrow & \quad \{\text{reflexão-}\times\} \\
 & id \cdot ([g, h] \times f) \\
\Leftrightarrow & \quad \{\text{definição de } id\} \\
 & [g, h] \times f
\end{array}
$$

III – Condicional

23. Partindo da definição do *combinador condicional* de McCarthy e da propriedade

$$p? \cdot f \quad = \quad (f + f) \cdot (p \cdot f)?$$

prove a validade de

$$(p \to f, g) \cdot h \quad = \quad (p \cdot h) \to (f \cdot h), (g \cdot h)$$

Resolução:

$$
\begin{array}{ll}
 & (p \to f, g) \cdot h \\
\Leftrightarrow & \quad \{\text{definição do combinador condicional de McCarthy}\} \\
 & ([f, g] \cdot p?) \cdot h \\
\Leftrightarrow & \quad \{\text{composição é associativa}\} \\
 & [f, g] \cdot (p? \cdot h) \\
\Leftrightarrow & \quad \{\text{propriedade dada acima }\} \\
 & [f, g] \cdot (h + h) \cdot (p \cdot h)? \\
\Leftrightarrow & \quad \{\text{absorção-+}\} \\
 & [f \cdot h, g \cdot h] \cdot (p \cdot h)? \\
\Leftrightarrow & \quad \{\text{definição do combinador condicional de McCarthy}\} \\
 & (p \cdot h) \to (f \cdot h), (g \cdot h)
\end{array}
$$

IV – Exponenciais

28. Demonstre as leis de *fusão* e *reflexão* da exponenciação, partindo da propriedade universal respectiva.

Resolução:
(parcial)

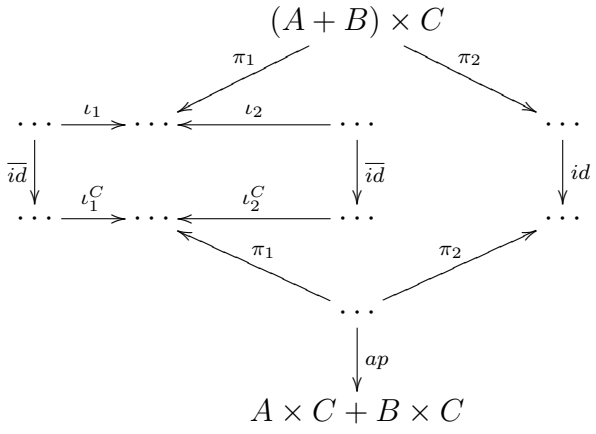
Partindo da propriedade universal da exponenciação prova-se a lei de fusão-exponencial

$$
\begin{array}{ll}
 & k = \overline{f} \Leftrightarrow f = ap \cdot (k \times id) \\
\Leftrightarrow & \quad \{\text{seja } k = \overline{g} \cdot f\} \\
 & \overline{g} \cdot f = \overline{f} \Leftrightarrow f = ap \cdot ((\overline{g} \cdot f) \times id) \\
\Leftrightarrow & \quad \{\text{substituição de } f\} \\
 & \overline{g} \cdot f = \overline{ap \cdot ((\overline{g} \cdot f) \times id)} \\
\Leftrightarrow & \quad \{\text{Nat-}id\} \\
 & \overline{g} \cdot f = \overline{ap \cdot ((\overline{g} \cdot f) \times (id \cdot id))} \\
\Leftrightarrow & \quad \{\text{functor-}\times\} \\
 & \overline{g} \cdot f = \overline{ap \cdot ((\overline{g} \times id) \cdot (f \times id))} \\
\Leftrightarrow & \quad \{\text{cancelamento-exponencial}\} \\
 & \overline{g} \cdot f = \overline{g \cdot (f \times id)}
\end{array}
$$

29. A função $distl : (A + B) \times C \to A \times C + B \times C$ (distributividade à esquerda) dispõe de uma definição *point-free* particularmente complicada:

$$distl = ap \cdot ([i_1^C \cdot \overline{id}, i_2^C \cdot \overline{id}] \times id)$$

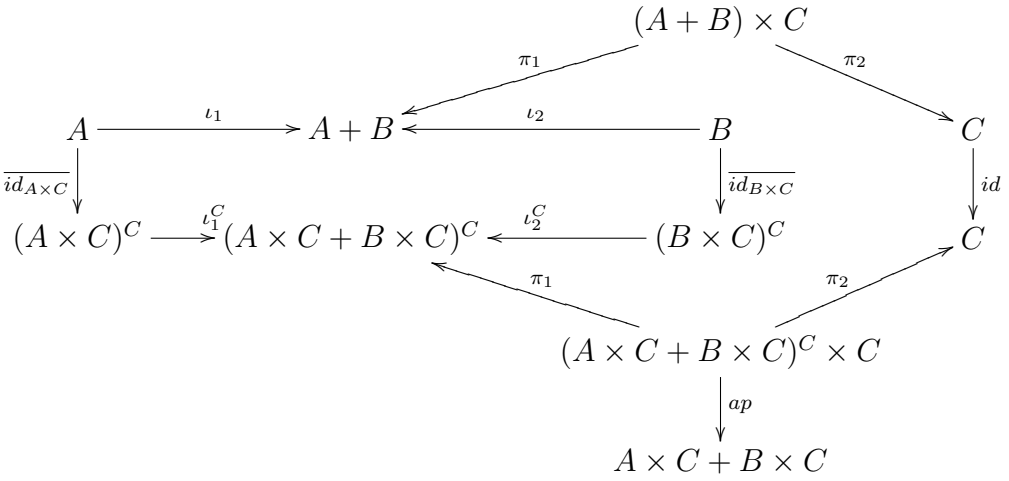
(a) Preencha o diagrama seguinte que justifica o tipo atribuído à função.



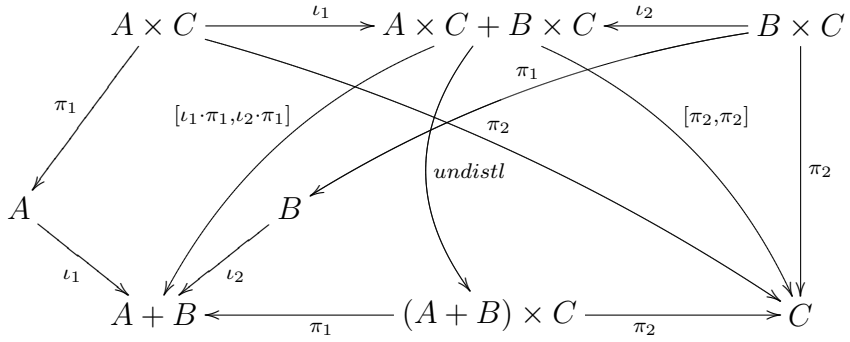
(b) Conjecture a inversa *undistl* e apresente diagramas que justifiquem o seu tipo.

Resolução:

(a)



(b)



Como demonstra o diagrama $undistl = \langle[\iota_1 \cdot \pi_1, \iota_2 \cdot \pi_1], [\pi_2, \pi_2]\rangle$, ou aplicando a lei da troca $undistl = [\langle\iota_1 \cdot \pi_1, \pi_2\rangle, \langle\iota_2 \cdot \pi_1, \pi_2\rangle]$.

V – *Point-free* na definição de funções

Capítulo 2

Cálculo Recursivo

I – Catamorfismos

5. No decorrer deste curso foi explicitada a relação entre a função `foldr` existente no prelúdio do HASKELL e os *catamorfismos* das listas. Em analogia com a função `foldr` das listas, considere a seguinte assinatura de função:

```
foldrX :: (String -> a -> a -> a) -> (Int -> a) -> X -> a
```

   Conjecture um tipo de dados `X` para o qual a função `foldrX` possa ser entendida de forma análoga a `foldr` nas listas.

   (a) Defina a função `foldrX` para o tipo de dados por si escolhido.

   (b) Considere as funções seguintes,

```
f = foldrX (\x y z -> 1+y+z) (\x -> 0)
g = foldrX (\x y z -> y ++ x ++ z) (\x -> (show x))
```

   Qual o tipo de cada uma das funções apresentadas? Diga resumidamente o que faz e para que serve cada uma. Dê exemplos de valores `x` e `y` de tal forma que `(f x)=2` e `(g y)="5 H 7"`.

   (c) Diga como poderia utilizar o tipo de dados `X` por si definido para representar funções aritméticas simples. Como representaria a expressão `5+(3*2)`?

   (d) Utilize agora a função `foldrX` para realizar a função de cálculo de expressões.

   Resolução:
   Inspeccionando `foldrX` infere-se que o tipo de dados `X` deverá constar de duas cláusulas, uma das quais (a não-recursiva) terá que ser um inteiro, e a outra das quais deverá envolver `String` e duas cópias de `X`. Assim, propõe-se:

```
data X = I Int | C String X X deriving (Eq,Show)
```

A função de *fold* associada a X será, então,

```
foldrX :: (String -> a -> a -> a) -> (Int -> a) -> X -> a
foldrX f g (I i) = g i
foldrX f g (C s e d) = f s (foldrX f g e) (foldrX f g d)
```

A função

```
f :: X -> Int
```

conta o número de *nodos intermédios* da árvore considerada. A função

```
g :: X -> String
```

concatena todos os valores contidos na árvore (as *strings* dos nodos intermédios e a respresentação textual dos inteiros contidos nas folhas, obtida por aplicação de `show`) de forma ordenada (cfr. travessia *in-order*).

Exemplos pedidos:

```
x = (C "x" (C "y" (I 1) (I 2)) (I 3))
y = (C " H " (I 5) (I 7))
```

Para representar expressões matemáticas no tipo `X` definido bastaria associar às operações binárias consideradas uma `String`, por exemplo associar à operação a `String` composta pelo seu símbolo.

Função de cálculo:

```
calc :: X -> Int
calc = foldrX calcAux id
       where calcAux s i1 i2 | s=="+" = i1 + i2
                             | s=="-" = i1 - i2
                             | s=="*" = i1 * i2
                             | s=="/" = i1 `div` i2
```

11. Antes de resolver as duas alíneas desta questão analize com atenção a seguinte arquitectura para a função

$$mdc \quad = \quad mul \cdot fpc \cdot (fp \times fp)$$

que calcula o máximo divisor comum entre dois números naturais, conforme o diagrama

$$\mathbf{N} \times \mathbf{N} \xrightarrow{fp \times fp} \mathbf{N}^* \times \mathbf{N}^* \xrightarrow{fpc} \mathbf{N}^* \xrightarrow{mul} \mathbf{N}$$

onde

- *fp* calcula os factores primos de um número listados por ordem crescente, por exemplo $fp\ 60 = [2, 2, 3, 5]$ e $fp\ 42 = [2, 3, 7]$;
- *fpc* intersecta duas listas de factores primos (*fpc* abrevia *factores primos comuns*), por exemplo $fpc\ ([2, 2, 3, 5], [2, 3, 7]) = [2, 3]$;
- *mul* multiplica os factores da lista produzida por *fpc*, inferindo assim o máximo divisor comum – $mdc\ (60, 42) = 2 * 3 = 6$.

(a) Complete a seguinte definição, em Haskell, da função

```
fpc' [] r = .........
fpc' l [] = .........
fpc' (a:l) (b:r) | a == b = .........
                 | a <  b = .........
                 | a >  b = .........
```

que é a versão *curried* de *fpc* (isto é, $fpc' = \overline{fpc}$):

(b) Escreva *mul* como um catamorfismo (de listas).

Resolução:

(a) Tem-se:

```
fpc' [] r = []
fpc' l [] = []
fpc' (a:l) (b:r)  | a == b = a : (fpc' l r)
                  | a <  b = fpc' l (b:r)
                  | a >  b = fpc' (a:l) r
```

(b) *mul* é o catamorfismo bem conhecido que multiplica todos os elementos de uma sequência: $mul = (\!|[\underline{1}, *]|\!)$.

II – Anamorfismos

III – Outros Exercícios sobre Anas e Catas

36. Considere o tipo de dados indutivo das *listas não vazias*:

```
data NRList a = Sing a | Add  (a , NRList a)
```

(a) Comece por definir `inNRList`, `outNRList`, `recNRList`, `cataNRList` e `anaNRList`.

(b) Dada a função `f = (const ()) -|- id`, qual é o tipo de `(inRList . f)`, onde `inRList` é uma função que conhece do módulo `RList.hs`? Desenhe os diagramas justificativos da sua resposta.

(c) Desenhe o diagrama do catamorfismo `cataNRList (inRList . f)` e calcule a função recursiva definida por este catamorfismo ao nível da variável, explicando o que a função obtida faz.

(d) Defina como catamorfismos as funções `maxim` e `minim` que calculam, respectivamente, o maior e o menor elemento de uma lista não vazia.

(e) Defina como um catamorfismo a função `maxmin :: NRList a -> (a,a)` que calcula o maior e o menor elementos de uma lista não vazia.

(f) Defina como um anamorfismo de `NRList` a função

```
inits :: [a] -> NRList [a]
```

que calcula os segmentos iniciais de uma lista.

Resolução:
(incompleta)

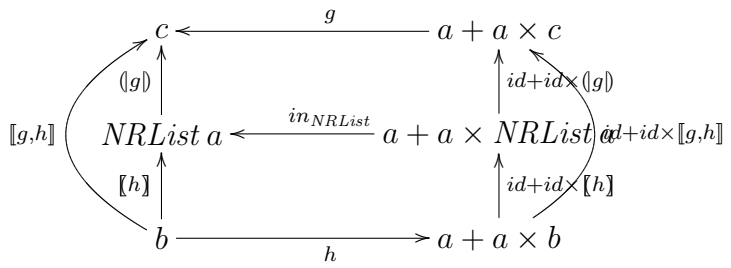
(a) O tipo $NRList\, a$ é definido pela equação

$$NRList\, a \underset{in_{NRList}}{\overset{out_{NRList}}{\equiv}} a + a \times NRList\, a$$

A base do tipo $NRList\, a$ é $B(a, x) = a + a \times x$. Logo, $rec\, f = B(id, f) = id + id \times f$. Assim, os seus diagramas *cata* e *ana* são, respectivamente,

$$\begin{array}{ccc} NRList\, a & \xrightarrow{out_{NRList}} & a + a \times NRList\, a \\ {\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|g|\!)} \\ c & \xleftarrow[g]{} & a + a \times c \end{array} \qquad \begin{array}{ccc} NRList\, a & \xleftarrow{in_{NRList}} & a + a \times NRList\, a \\ {\scriptstyle [\!(h)\!]}\uparrow & & \uparrow{\scriptstyle id+id\times[\!(h)\!]} \\ b & \xrightarrow[h]{} & a + a \times b \end{array}$$

que, combinados entre si dão o diagrama *hilo*:



Já vimos que $rec\, f = B(id, f) = id + id \times f$. Assim:

```
inNRList = either Sing Add
outNRList (Sing x) = Left x
outNRList (Add (x,l)) = Right (x,l)
rec f = id -|- (id >< f)
cata g = g . (rec (cata g)) . outNRList
ana h = inNRList . (rec (ana h) ) . h
hylo g h = (cata g) . (ana h)
```

(d)
```
maxim :: Ord a => NeList a -> a
maxim = cata (either id (uncurry max))

minim :: Ord a => NeList a -> a
minim = cata (either id (uncurry min))
```

(e)
```
maxmin :: Ord a => NeList a -> (a,a)
maxmin = cata (either (split id id) f)
         where f (a,(x,y)) = ((max a x),(min a y))
```

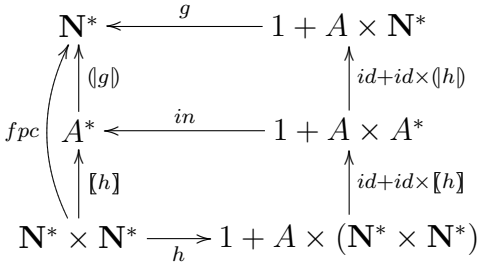
(f) Por segmento inicial de uma lista entende-se um seu prefixo, por exemplo, $[u, m]$ é prefixo de $[u, m, a]$. Qualquer sequência é prefixo dela própria e a sequência vazia é prefixo de qualquer outra. Na proposta que se segue, `reverse` é a função definida no Prelúdio do Haskell que inverte uma lista, e `blast` deve ler-se *all but last*:

```
inits :: [a] -> NeList [a]
inits = ana f
    where f [] = Left []
          f l  = Right (l, blast l)
                 where blast = reverse.tail.reverse
```

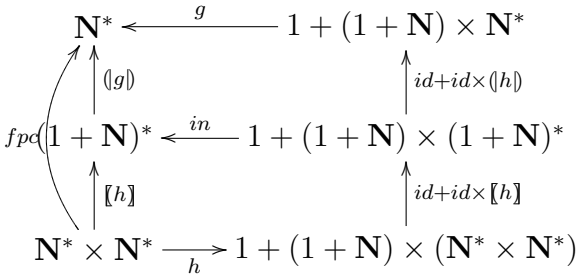
IV – Hilomorfismos

46. Escreva a função *fpc* do Ex. 11 como um hilomorfismo.
Resolução:

Exprimir fpc como um hilomorfismo de listas implica encontrar A, g e h do diagrama que se segue:



Devemos pensar em encontrar A primeiro que tudo. As três primeiras cláusulas de fpc' sugerem $A = \mathbf{N}$, mas para $a < b$ e $a > b$ não existe nenhum valor do tipo A para construir a lista A^* intermédia, o que sugere $A = 1 + \mathbf{N}$. Assim, o diagrama fica:



Podemos agora decalcar h a partir de fpc', usando `Maybe` A para representar $1 + A$:

```
h([],r) = i1()
h(l,[]) = i1()
h(a:l,b:r) | a == b = i2(Just a,(l,r))
           | a <  b = i2(Nothing,(l,b:r))
           | a >  b = i2(Nothing,(a:l,r))
```

Quanto a g, será sempre da forma $[\underline{[]}, g']$ onde g' apenas tem que ignorar os valores nulos (`Nothing`):

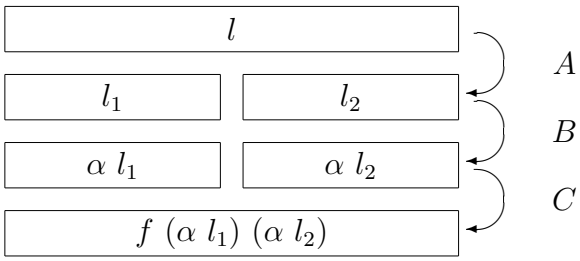
```
g'(Just a,l)  = a:l
g'(Nothing,l) = l
```

Se recorrermos ao módulo `RList`, teremos

```
fpc =  hyloRList (either (const []) g') h
```

V– Hilomorfismos e Classes Algorítmicas *Standard*

62. O seguinte desenho pretende descrever graficamente um algoritmo α de ordenação de listas bem conhecido:



(a) Identifique α, bem como as suas fases A, B e C e a função f. Codifique esta última em Haskell.

(b) Descreva o mesmo algoritmo sob a forma de um hilomorfismo de um particular tipo indutivo estudado nas aulas desta disciplina.

Resolução:

(a) $\alpha =$ `mSort` (*merge sort*) e f é a função `merge` da biblioteca `LTree.hs`.
Fases do algoritmo: A — análise da lista argumento e sua representação em duas sublistas de igual (ou quase igual) comprimento; B — passo recursivo, i.e. ordenação das duas sublistas construídas em A; C — síntese do resultado por fusão de duas listas ordenadas.

(b) Ver `mSort` em `LTree.hs`.
O hilomorfismo implícito em *mSort* exclui listas não-vazias, que estão trivialmente ordenadas. Listas singulares também estão trivialmente ordenadas e como tal não oferecem problemas. Na fase A, uma função $sep : A^* \to A^* \times A^*$ vai partir listas não singulares em pares de listas não vazias, sem *pivot* (é esta a principal diferença entre o *quick sort* e o *merge sort*). Isto sugere, em conjunto com o caso singular, a seguinte estrutura de dados intermédia para o hilomorfismo:

$$LTree\ A \quad \equiv \quad A + LTree\ A \times LTree\ A$$

conhecida pelo nome de *árvore com folhas*. Surge assim o anamorfismo *de separação*

$$\begin{array}{ccc} LTree\ A & \xleftarrow{in} & A + LTree\ A \times LTree\ A \\ {\scriptstyle [\![h]\!]}\uparrow & & \uparrow{\scriptstyle id+[\![h]\!]\times[\![h]\!]} \\ A^* & \xrightarrow[h]{} & A + A^* \times A^* \end{array}$$

onde h (designada *lsplit* em `LTree.hs`) tem o propósito de bilinearizar o algoritmo (tal como o seu equivalente no *quick sort*). A estrutura de dados intermédia será agora consumida por um catamorfismo baseado na função $merge : A^* \times A^* \rightarrow A^*$ que funde listas ordenadas. Basta construir $(\!|g|\!)$ para $g = [singl, merge]$ e adicioná-lo ao diagrama:

$$\begin{array}{ccc} A^* & \xleftarrow{g} & A + A^* \times A^* \\ {\scriptstyle (\!|g|\!)}\uparrow & & \uparrow{\scriptstyle id+(\!|g|\!)\times(\!|g|\!)} \\ LTree\ A & \xleftarrow{in} & A + LTree\ A \times LTree\ A \\ {\scriptstyle [\![h]\!]}\uparrow & & \uparrow{\scriptstyle id+[\![h]\!]\times[\![h]\!]} \\ A^* & \xrightarrow[h]{} & A + A^* \times A^* \end{array}$$

VI – Cálculo com Funções Recursivas

67. A igualdade que se segue é conhecida pelo nome de *banana-split* e traduz uma permutatividade célebre entre *splits* e catamorfismos.

$$\langle (\!|g|\!), (\!|k|\!) \rangle = (\!| \langle g \cdot \mathsf{F}\,\pi_1, k \cdot \mathsf{F}\,\pi_2 \rangle |\!)$$

(a) Verifique que ambos os membros da igualdade exibem o mesmo tipo.

(b) Mostre ainda que a lei se pode escrever da forma alternativa seguinte:

$$\langle (\!|g|\!), (\!|k|\!) \rangle = (\!| (g \times k) \cdot \langle \mathsf{F}\,\pi_1, \mathsf{F}\,\pi_2 \rangle |\!)$$

(c) A função que calcula a média dos elementos de listas de números naturais

$$media \quad = \quad div \cdot \langle soma, comp \rangle$$

combina dois catamorfismos: $soma = (\!| [\underline{0}, add] |\!)$ e $comp = (\!| [\underline{0}, succ \cdot \pi_2] |\!)$. Aplique à função *media* a lei anterior e traduza a função resultado para a notação com variáveis. Qual é a vantagem desta última em termos de eficiência?

Resolução:
Tomemos então $\mathsf{F} f = id + id \times f$ no contexto de listas: $\mathsf{L}\,a \equiv 1 + a \times \mathsf{L}\,a$.

O catamorfismo $(\!|g|\!)$ será de tipo $: \mathsf{L}\,A \to B$ desde que g seja de tipo $g : 1 + A \times B \leftarrow B$. O catamorfismo $(\!|k|\!)$ terá que ter o mesmo domínio $\mathsf{L}\,A$ pois está envolvido num *split* com $(\!|g|\!)$. Teremos $(\!|k|\!) : \mathsf{L}\,A \to C$ para $k : 1 + A \times C \leftarrow C$. Assim, o lado esquerdo da igualdade exibe tipo

$$\langle (\!|g|\!) , (\!|k|\!) \rangle : \mathsf{L}\,A \leftarrow B \times C$$

Vejamos o lado direito, escrito da forma alternativa $(\!| (g \times k) \cdot \langle F\,\pi_1, F\,\pi_2 \rangle |\!)$ permitida por aplicação da lei da absorção-$\times$ em sentido inverso. Pegando nos tipos de g e k inferidos do lado esquerdo, teremos $g \times h : (1 + A \times B) \times (1 + A \times C) \leftarrow B \times C$. Portanto, $F\,\pi_1 = id + id \times \pi_1$ deverá ter codomínio $1 + A \times B$ e $F\,\pi_2 = id + id \times \pi_2$ deverá ter codomínio $1 + A \times C$ e ambas as funções deverão ter o mesmo domínio por estarem no mesmo *split*. Ora basta que esse domínio seja $1 + A \times (B \times C)$, isto é,

$$(g \times k) \cdot \langle F\,\pi_1, F\,\pi_2 \rangle : 1 + A \times (B \times C) \leftarrow B \times C$$

cujo catamorfismo é de tipo $: \mathsf{L}\,A \leftarrow B \times C$ compatível, portanto, com o do lado esquerdo da igualdade.

Seja $f = \langle soma, comp \rangle$, isto é $f = \langle (\!| [\underline{0}, add] |\!) , (\!| [\underline{0}, succ \cdot \pi_2] |\!) \rangle$, tendo-se assim $media = div \cdot f$. O nosso objectivo é transformar f num catamorfismo:

$$f = \langle (\!| \underbrace{[\underline{0}, add]}_{g} |\!), (\!| \underbrace{[\underline{0}, succ \cdot \pi_2]}_{k} |\!) \rangle$$

$\Leftrightarrow$ {por *banana-split* }

$$f = (\!| \langle [\underline{0}, add] \cdot (id + id \times \pi_1), [\underline{0}, succ \cdot \pi_2] \cdot (id + id \times \pi_2) \rangle |\!)$$

$\Leftrightarrow$ {por *absorção-+* e constante $\underline{0}$ }

$$f = (\!| \langle [\underline{0}, add \cdot (id \times \pi_1)], [\underline{0}, succ \cdot \pi_2 \cdot \pi_2] \rangle |\!)$$

$\Leftrightarrow$ {pela lei da troca }

$$f = (\!| [\langle \underline{0}, \underline{0} \rangle, \langle add \cdot (id \times \pi_1), succ \cdot \pi_2 \cdot \pi_2 \rangle] |\!)$$

$\Leftrightarrow$ {definição de catamorfismo de listas, onde $in = [\underline{Nil}, Cons]$ e $rec\, f = id + id \times f$ }

$$f \cdot [\underline{Nil}, Cons] = [\langle \underline{0}, \underline{0} \rangle, \langle add \cdot (id \times \pi_1), succ \cdot \pi_2 \cdot \pi_2 \rangle] \cdot (id + id \times f)$$

$\Leftrightarrow$ {por *absorção-+* e identidade id }

$$f \cdot [\underline{Nil}, Cons] = [\langle \underline{0}, \underline{0} \rangle, \langle add \cdot (id \times \pi_1), succ \cdot \pi_2 \cdot \pi_2 \rangle \cdot (id \times f)]$$

$\Leftrightarrow$ {introdução de variáveis }

$$\begin{cases} (f \cdot \underline{Nil})x = \langle \underline{0}, \underline{0} \rangle\, x \\ (f \cdot Cons)(a, l) = (\langle add \cdot (id \times \pi_1), succ \cdot \pi_2 \cdot \pi_2 \rangle \cdot (id \times f))(a, l) \end{cases}$$

$\Leftrightarrow$ { funções constantes e definições de $f \times g$ e de id }

$$\begin{cases} f\, Nil = (0, 0) \\ f(Cons(a, l)) = \langle add \cdot (id \times \pi_1), succ \cdot \pi_2 \cdot \pi_2 \rangle (a, f\, l) \end{cases}$$

$\Leftrightarrow$ { fazendo $f\, l = (x, y)$ }

$$\begin{cases} f\, Nil = (0, 0) \\ f(Cons(a, l)) = \\ \qquad let\ (x, y) = f\, l \\ \qquad in\ \langle add \cdot (id \times \pi_1), succ \cdot \pi_2 \cdot \pi_2 \rangle (a, (x, y)) \end{cases}$$

$\Leftrightarrow$ { resolução do *split* e do $\times$ que restam e definições de π_1 e π_2 }

$$\begin{cases} f\, Nil = (0, 0) \\ f(Cons(a, l)) = \\ \qquad let\ (x, y) = f\, l \\ \qquad in\ (add(a, x), succ\, y) \end{cases}$$

Em Haskell:

```
import LList
media = div . f
where f Nil = (0,0)
      f (Cons(a,l)) = let (x,y)= f l in (add(a,x),succ y)
```

```
add(a,b)= a + b
div = uncurry (/)
```

A versão de f a que se chegou é mais eficiente por fazer apenas uma travessia da lista argumento, enquanto que a versão de que partimos, $\langle soma, comp \rangle$, faz duas: a de *soma* e a de *comp*.

VII – Outros Tópicos