

X - Tipos Indutivos (Cont); Introdução aos Functores

Estudo de um Caso: Listas

1. Uma álgebra de Listas

Ponto da situação: partindo de uma definição recursiva de um tipo de dados, escreveu-se a equação recursiva

$$L \cong 1 + A \times L$$

e “adivinhou-se” uma solução possível, A^*

$$A^* \cong 1 + A \times A^*$$

$$A^* \cong 1 + A \times A^*$$

Colocam-se duas questões:

- Qual a álgebra que testemunha este isomorfismo?
- Será A^* a única solução para a equação?

Seja $in = [in_1, in_2]$ essa álgebra:

$$\begin{array}{ccccc} 1 & \xrightarrow{i_1} & 1 + A \times A^* & \xleftarrow{i_2} & A \times A^* \\ & \searrow in_1 & \downarrow in & \swarrow in_2 & \\ & & A^* & & \end{array}$$

Situamo-nos num domínio abstracto de sequências finitas, e não numa linguagem de programação particular. Sejam nesse domínio $[]$ a sequência vazia, e um operador

$$\begin{aligned} cons : A \times A^* &\longrightarrow A^* \\ cons(a, [a_1, \dots, a_n]) &= [a, a_1, \dots, a_n] \end{aligned}$$

Temos imediatamente uma definição para *in*:

$$\begin{array}{ccccc}
 1 & \xrightarrow{i_1} & 1 + A \times A^* & \xleftarrow{i_2} & A \times A^* \\
 & \searrow & \downarrow & & \swarrow \\
 & \underline{[]} & & & \underline{[\]}, cons \\
 & & & & cons
 \end{array}$$

e também *out* $\stackrel{\text{def}}{=} (! + \langle hd, tl \rangle) \bullet (=_{[]} ?)$ com

$$hd : A^* \longrightarrow A$$

$$hd [a_1, a_2, \dots, a_n] = a_1$$

$$tl : A^* \longrightarrow A^*$$

$$tl [a_1, a_2, \dots, a_n] = [a_2, \dots, a_n]$$

2- Soluções Isomorfas

Relembremos que a equação que estudamos,

$$L \cong 1 + A \times L$$

nasceu de uma definição de um tipo de dados
(em duas linguagens distintas). Em Haskell:

```
data T = Nil | Cons (A, T)
```

O próprio tipo de dados T constitui uma solução para a equação, com a álgebra $T \xleftarrow{in_T} 1 + A \times T$

$$in_T = [\underline{Nil}, Cons]$$

in_T denota pois a álgebra associada ao tipo T .

A co-álgebra de T pode ser escrita como

$$out_T : T \longrightarrow 1 + A \times T$$

$$out_T Nil = i_1 NIL$$

$$out_T(Cons(a, l)) = i_2(a, l)$$

Qualquer tipo de dados resultante apenas de uma mudança de nome dos construtores, como

```
data U = Stop | Join (A, U)
```

é também solução da mesma equação, correspondendo-lhe uma álgebra própria, neste caso [Stop, Join]

Existe pois toda uma família de tipos de dados *isomorfos* que são solução da equação inicial

$$L \cong 1 + A \times L$$

O que significa dizer que as listas constituem um tipo de dados indutivo?

Simplesmente que a sua *definição* é indutiva:

- *Nil* é uma lista (**caso de base**)
- Se $x : A$, e l é uma lista, então $Cons(x, l)$ é também uma lista (**caso indutivo**)

3- Definição de Functor

Um functor é um mapeamento de tipos
(uma forma de construir tipos a partir de outros)

com uma particularidade:
deve também ser capaz de mapear funções

Seja F um functor e A e B tipos;

Seja ainda f uma função $B \xleftarrow{f} A$. Então:

- $F A$ e $F B$ são ainda tipos

- $F f$ é uma função $F B \xleftarrow{F f} F A$

$$\begin{array}{ccc} A & \cdots\cdots\cdots & F A \\ f \downarrow & & \downarrow F f \\ B & \cdots\cdots\cdots & F B \end{array}$$

Para que F seja de facto um functor, é ainda necessário que duas condições sejam cumpridas:

- F mapeia a identidade na identidade
- F comuta com a composição de funções

$$F id_A = id_{(FA)}$$

$$F(g \bullet h) = (Fg) \bullet (Fh)$$

4- Functores de Tipo

Qualquer construtor de tipo de dados polimórfico é um functor. Por exemplo o functor List:

```
data List a = Nil | Cons a (List a)
```

Mapeia um tipo X em List X , e uma função f na função List $f = map f$

Exercício: provar que List é de facto um functor

XI - Functores e Tipos

1- Functor Exponenciação

Para um determinado tipo A fixo, tem-se:

$$\mathsf{F} X \stackrel{\text{def}}{=} X^A$$

$$\mathsf{F} f \stackrel{\text{def}}{=} \overline{f \bullet ap}$$

Compreende-se agora a designação das
“Propriedades Functoriais” da exponenciação...

2- Functores Elementares

- Functor Identidade:

$$\mathsf{F} X = X; \quad \mathsf{F} f = f$$

- Functor Constante:

$$\mathsf{F} X = C; \quad \mathsf{F} f = id_C$$

Exercício: provar que são de facto functores

3- Bifunctores

Um bifunctor \mathbf{B} mapeia

- um *par de tipos* X, Y num novo tipo
- um *par de funções* $f : X \rightarrow U, g : Y \rightarrow V$ numa função $\mathbf{B}(f, g) : \mathbf{B}(X, Y) \rightarrow \mathbf{B}(U, V)$

tendo que se verificar

$$\mathbf{B}(id_A, id_B) = id_{\mathbf{B}(A, B)}$$

$$\mathbf{B}(g \bullet h, i \bullet j) = \mathbf{B}(g, i) \bullet \mathbf{B}(h, j)$$

Usando notação infixa \odot para o bifunctor B , teríamos

$$\text{id}_A \odot \text{id}_B = \text{id}_{A \odot B}$$

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

4- Bifunctores Produto e Coproduto

As propriedades functoriais anteriores tornam evidente que

- os constructores de tipos **produto** e **coproduto** são bifunctores, em que
- o mapeamento de funções é dado pelos combinadores **produto** e **soma**, respectivamente.

$$A \times B \xleftarrow{f \times g} C \times D$$

$$A + B \xleftarrow{f+g} C + D$$

Mais uma vez se comprehende a designação das
“Propriedades Functoriais” destes combinadores

5- Produto e Coproduto de Functores

Os bifunctores produto e coproduto podem ser *lifted* por forma a permitir construir novos functores:

$$(F + G) X = F X + G X \quad (F \times G) X = F X \times G X$$

$$(F + G) f = F f + G f \quad (F \times G) f = F f \times G f$$

Note-se que $+$ e \times designam entidades diferentes...

Exercício: provar que são ainda functores...

6- Composição de Functores

Definição óbvia:

$$(F \bullet G)X \stackrel{\text{def}}{=} F(G X)$$

$$(F \bullet G)f \stackrel{\text{def}}{=} F(G f)$$

7- Functores Polinomiais

Um functor polinomial define-se como sendo:

- um functor constante, ou
- o functor identidade, ou
- o produto ou coproduto de dois functores polinomiais, ou
- a composição de dois functores polinomiais

É fácil, dada a definição de um functor polinomial nos tipos, chegar à sua definição nas funções:

$$\mathsf{F} X \stackrel{\text{def}}{=} 1 + A \times X$$

$$\mathsf{F} f \stackrel{\text{def}}{=} \mathsf{id} + \mathsf{id} \times f$$

Como se generaliza este esquema?

8 - Normalização

Os functores polinomiais podem ser normalizados,
de forma idêntica aos polinómios.

$$F X = A \times (1 + X)^2$$

$$\underbrace{A}_{C_0} + \underbrace{A \times 2}_{C_1} \times X + \underbrace{A}_{C_2} \times X^2$$

Exercício: obter a forma normal acima

9- Tipos Indutivos Polinomiais

Tipo $T = \mu F$ gerados por um functor polinomial F :

$$T \underset{in_T}{\curvearrowleft} \cong \sum_{i=0}^n C_i \times T^i \quad in_T \stackrel{\text{def}}{=} [f_1, \dots, f_n]$$

```
data T = C0 |  
        C1 (C1, T) |  
        C2 (C2, (T, T)) |  
        ... |  
        Cn (Cn, (T, ..., T))
```

XII - Pontos Fixos de Functores Catamorfismos de Listas

1- Pontos Fixos e Tipos de Dados

Regressemos à equação recursiva das listas de A :

$$L \cong 1 + A \times L$$

esta pode ser reescrita como:

$$L \cong \mathsf{F} \ L$$

com

$$\mathsf{F} \ X \stackrel{\text{def}}{=} 1 + A \times X$$

$$L \cong F L$$

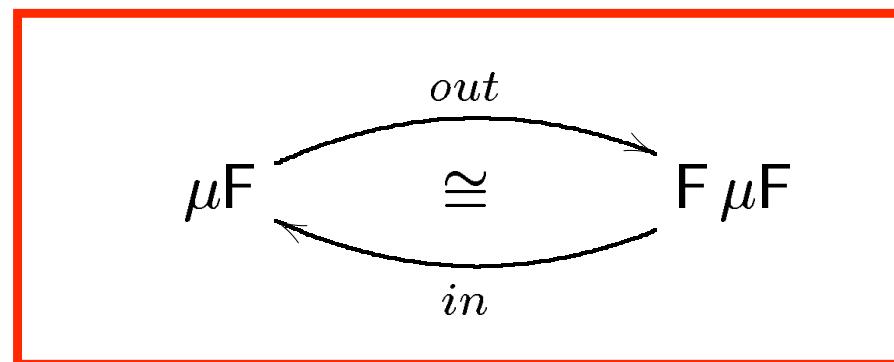
Vimos já que uma tal equação possui como soluções uma classe de tipos isomorfos.

Estes tipos são designados **Pontos Fixos** do functor F , que representaremos por μF .

Dizemos também que F é o **functor-base** do tipo μF .

por exemplo $F X \stackrel{\text{def}}{=} 1 + A \times X$

é o functor-base do tipo “listas de A”.



Exercício:

- Determinar os functores-base dos tipos LTree e BTree
- Como mapeiam estes uma função?

2- Catamorfismos de Listas

Seja T o tipo “Lista de Números Inteiros”:

$$T \xleftarrow{in_T} 1 + \mathbb{N}_0 \times T$$

Então a função “somatório” pode ser definida sem variáveis como

$$f = [\underline{0}, add] \bullet (id + id \times f) \bullet out_T$$

$$\text{ou } f \bullet [\underline{Nil}, Cons] = [\underline{0}, add] \bullet (id + id \times f)$$

$$\begin{array}{ccc}
 T & \xrightarrow{\quad out_T \quad} & 1 + \mathbb{N}_0 \times T \\
 f \downarrow & & \downarrow id + id \times f \\
 \mathbb{N}_0 & \xleftarrow{[\underline{0}, add]} & 1 + \mathbb{N}_0 \times \mathbb{N}_0
 \end{array}$$

$$\begin{cases} f Nil = 0 \\ f(Cons(a, x)) = a + f x \end{cases}$$

Ex: efectuar esta conversão para código com variáveis

Claramente a abstracção da função $[\underline{0}, add]$ leva-nos a uma definição *point-free* do conhecido padrão de recursividade *foldr* das listas.

$$\begin{array}{ccc}
 T & \xrightarrow{\quad out_T \quad} & 1 + \mathbb{N}_0 \times T \\
 f \downarrow & & \downarrow id + id \times f \\
 B & \xleftarrow{\quad g \quad} & 1 + \mathbb{N}_0 \times B
 \end{array}$$

Chamaremos a f o *catamorfismo* gerado por g , e escreveremos $f = ([g])$

$$[g] = g \bullet (id + id \times [g]) \bullet out_T$$

Exercícios: definir os catamorfismos

- Comprimento de uma lista
- Soma (dos elementos) de uma lista
- *map* de listas

XIII - Catamorfismos

1- Catamorfismos Genéricos

Generalização da noção anterior, para um tipo polinomial genérico com functor-base F

$$\begin{array}{ccc} \mu F & \begin{matrix} \xrightarrow{\quad out \quad} \\ \cong \\ \xleftarrow{\quad in \quad} \end{matrix} & F \mu F \\ & & \\ & f = ([\alpha])_F & \downarrow \\ & & A \xleftarrow[\alpha]{} FA \\ & & \downarrow \\ & \mu F & \xleftarrow{in} F \mu F \\ & & \downarrow \\ & & F([\alpha])_F \end{array}$$

$$([\alpha]) = \alpha \cdot F([\alpha]) \cdot out$$

2- Propriedade Universal

$$\begin{array}{ccc} \mu F & \xleftarrow{in} & F \mu F \\ f = ([\alpha])_F \downarrow & & \downarrow F([\alpha])_F \\ A & \xleftarrow[\alpha]{} & FA \end{array}$$

$$k = [\alpha] \Leftrightarrow k \cdot in = \alpha \cdot F k$$

3- Props: Cancelamento

$$([\alpha]) \bullet in = \alpha \bullet F([\alpha])$$
$$\begin{array}{ccc} \mu F & \xleftarrow{in} & F \mu F \\ f = ([\alpha])_F \downarrow & & \downarrow F([\alpha])_F \\ A & \xleftarrow[\alpha]{} & FA \end{array}$$

4- Props: Fusão

Para mais tarde...

5- Props: Reflexão

$$\begin{array}{ccc} \mu F & \xleftarrow{\quad in \quad} & F \mu F \\ ([in]) \downarrow & & \downarrow F([in]) \\ \mu F & \xleftarrow{\quad in \quad} & F \mu F \end{array}$$

Prova (utilizzando propr. univ.)

$$\begin{aligned} id = [\alpha] &\Leftrightarrow id \bullet in = \alpha \bullet F id \\ &= \{ \text{ identity (1.10) and } F \text{ is a functor (2.44) } \} \\ id = [\alpha] &\Leftrightarrow in = \alpha \bullet id \\ &= \{ \text{ identity (1.10) once again } \} \\ id = [\alpha] &\Leftrightarrow in = \alpha \\ &= \{ \alpha \text{ replaced by } in \text{ and simplifying } \} \\ id = [in] & \end{aligned}$$

Observe-se que

- Trata-se de uma definição *única* de catamorfismo, parametrizada pelo tipo indutivo em questão
- Introduz-se no cálculo um conjunto de leis, que permitirão raciocinar sobre os *folds* de qualquer tipo de dados, da mesma forma que *split*, *either*, *curry*, ...

6- Conceitos Associados

- A álgebra *in* de um tipo de dados é designada de *inicial*, o que é justificado pela propr. univ. dos catamorfismos.
- Qualquer outra função $A \xleftarrow{\alpha} \mathbf{F} A$ é uma álgebra do mesmo functor
- É tipicamente gerada por uma declaração `data ...`

Exercícios: instanciar as leis dos catamorfismos para os tipos

- LTree
- BTree
- RTree

XIV - Anamorfismos

1- Anamorfismos de Listas

Voltemos ao tipo $T = \text{“Lista de Números Inteiros”}$:

$$T \xleftarrow{in_T} 1 + \mathbb{N}_0 \times T$$

Então a função “downto” pode ser definida sem variáveis como

$$f = in_T \cdot (id + id \times f) \cdot (! + \langle id, \check{z} `` . \rangle) \cdot f\# `` \check{z} ?$$

A **dualização** da noção de catamorfismo leva-nos a uma nova **definição point-free**, de um **padrão de recursividade** das listas.

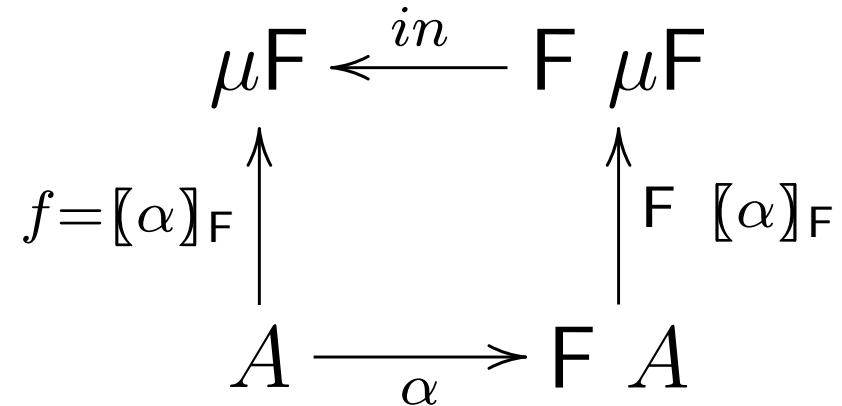
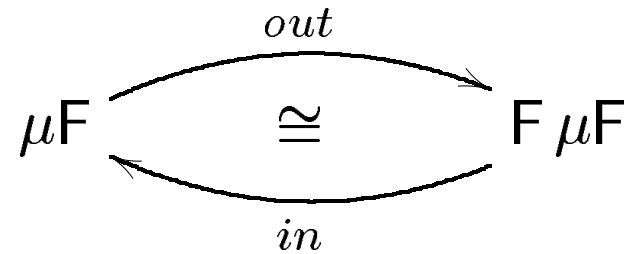
$$\begin{array}{ccc} T & \xleftarrow{\quad in_T \quad} & 1 + \mathbb{N}_0 \times T \\ \uparrow f & & \uparrow id + id \times f \\ B & \xrightarrow{\quad g \quad} & 1 + \mathbb{N}_0 \times B \end{array}$$

Chamaremos a f o *anamorfismo* gerado por g , e escreveremos $f = [(g)]$

$$[(g)] = in_T \bullet (id + id \times [(g)]) \bullet g$$

2- Anamorfismos Genéricos

Generalização desta noção, para um tipo polinomial genérico com functor-base F



$$[\alpha] = in \cdot F[\alpha] \cdot \alpha$$

3- Propriedade Universal

$$\begin{array}{ccc} \mu F & \xrightarrow{\text{out}} & F \mu F \\ \uparrow [\alpha]_F & & \uparrow F [\alpha]_F \\ A & \xrightarrow[\alpha]{} & F A \end{array}$$

$$k = [\alpha] \Leftrightarrow \text{out} \cdot k = F k \cdot \alpha$$

4- Props: Cancelamento

$$\begin{array}{ccc} \mu F & \xrightarrow{\text{out}} & F \mu F \\ \uparrow [\alpha]_F & & \uparrow F [\alpha]_F \\ A & \xrightarrow{\alpha} & F A \end{array}$$

$$\text{out} \cdot [\alpha] = F[\alpha] \cdot \alpha$$

5- Props: Fusão

Para mais tarde...

6- Props: Reflexão

$$[\text{out}] = \text{id}_{\mu F}$$

$$\begin{array}{ccc} \mu F & \xrightarrow{\text{out}} & F \mu F \\ \uparrow [\text{out}]_F & & \uparrow F [\text{out}]_F \\ \mu F & \xrightarrow{\text{out}} & F \mu F \end{array}$$

Observe-se que

- Trata-se de uma definição *única* de anamorfismo, parametrizada pelo tipo indutivo em questão
- Introduz-se no cálculo um conjunto de leis, que permitirão raciocinar sobre os *unfolds* de qualquer tipo de dados, da mesma forma que *split*, *either*, *curry*, *folds*...

Exercícios:

outros exemplos de anamorfismos de listas: zip

instanciar as leis dos anamorfismos para os tipos

- LTree
- BTree

exemplos:

- construção de uma árvore de pesquisa
- construção de uma LTree equilibrada

XV - Hilomorfismos

1- Hilomorfismos

Dados:

- um tipo indutivo gerado por um functor-base F ;
- um anamorfismo a de F ,
que **gera** uma estrutura do tipo μF ;
- um catamorfismo c de F ,
que **consome** uma estrutura do tipo μF ;

Chamaremos à composição $h = c . a$ um
hilomorfismo de F .

2- Hilomorfismos de Listas de Inteiros

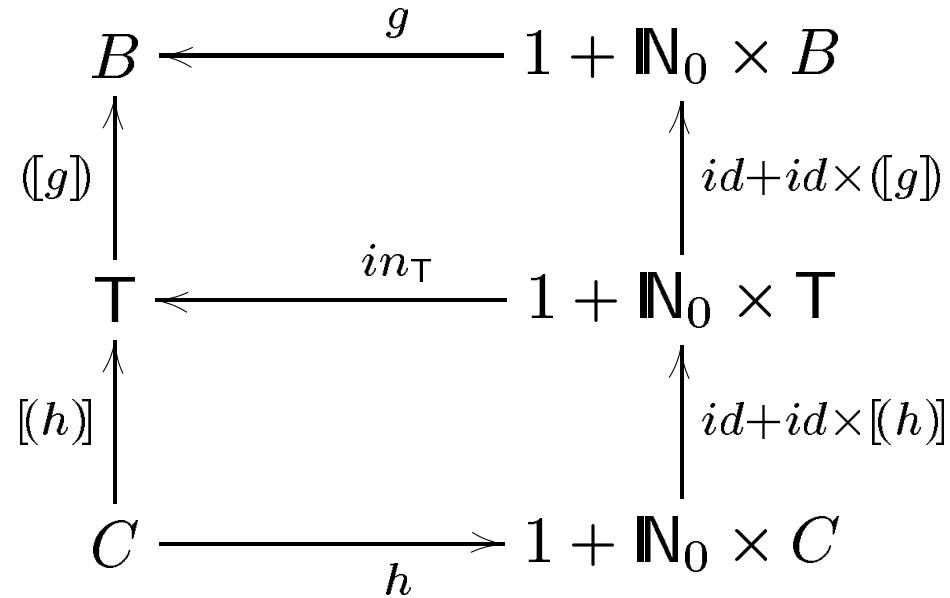
Dados os genes

$$C \xrightarrow{h} 1 + \mathbb{N}_0 \times C \quad B \xleftarrow{g} 1 + \mathbb{N}_0 \times B$$

que definem as funções

$$B \xleftarrow{([g])} T \xleftarrow{[(h)]} C$$

T é o *tipo intermédio* do hilomorfismo $([g]) \bullet [(h)]$



Poder-se-á definir directamente $([g]) \bullet ([h])$
sem construir a estrutura de dados intermédia?

$$\begin{aligned}
& \llbracket g \rrbracket \bullet \llbracket h \rrbracket = g \bullet (id + id \times \llbracket g \rrbracket) \bullet (id + id \times \llbracket h \rrbracket) \bullet h \\
\Leftrightarrow & \quad \{ \text{ +-functor (1.42) } \} \\
& \llbracket g \rrbracket \bullet \llbracket h \rrbracket = g \bullet ((id \bullet id) + (id \times \llbracket g \rrbracket) \bullet (id \times \llbracket h \rrbracket)) \bullet h \\
\Leftrightarrow & \quad \{ \text{ identity and } \times\text{-functor (1.28) } \} \\
& \llbracket g \rrbracket \bullet \llbracket h \rrbracket = g \bullet (id + id \times \llbracket g \rrbracket \bullet \llbracket h \rrbracket) \bullet h
\end{aligned}$$

$$\begin{array}{ccc}
B & \xleftarrow{g} & 1 + \mathbb{N}_0 \times B \\
\uparrow \llbracket g \rrbracket \bullet \llbracket h \rrbracket & & \uparrow id + id \times \llbracket g \rrbracket \bullet \llbracket h \rrbracket \\
C & \xrightarrow{h} & 1 + \mathbb{N}_0 \times C
\end{array}$$

3- Exemplo

Sejam

$$g = [\underline{1}, \text{mul}]$$

$$h = (! + \langle id, pred \rangle) \cdot (=_0 ?)$$

Calculemos:

$$[\![g]\!] \bullet [\![h]\!] = g \bullet (id + id \times [\![g]\!] \bullet [\![h]\!]) \bullet h$$

\leftrightarrow { $[\![g]\!] \bullet [\![h]\!]$ abbreviated to f and instantiating h and g }

$$f = [\underline{1}, \text{mul}] \bullet (id + id \times f) \bullet (! + \langle id, pred \rangle) \bullet (=_0 ?)$$

\leftrightarrow { +-functor (1.42) and identity }

$$f = [\underline{1}, \text{mul}] \bullet (! + (id \times f) \bullet \langle id, pred \rangle) \bullet (=_0 ?)$$

$$\begin{aligned}
 f &= [\underline{1}, \text{mul}] \bullet (! + (\text{id} \times f) \bullet \langle \text{id}, \text{pred} \rangle) \bullet (=_0 ?) \\
 \leftrightarrow &\quad \{ \times\text{-absorption (1.25) and identity} \} \\
 f &= [\underline{1}, \text{mul}] \bullet (! + \langle \text{id}, f \bullet \text{pred} \rangle) \bullet (=_0 ?) \\
 \leftrightarrow &\quad \{ +\text{-absorption (1.41) and constant } \underline{1} \text{ (1.15)} \} \\
 f &= [\underline{1}, \text{mul} \bullet \langle \text{id}, f \bullet \text{pred} \rangle] \bullet (=_0 ?)
 \end{aligned}$$

Introduzindo variáveis reconhecemos $f = \text{factorial}$:

$$\left\{
 \begin{array}{l}
 f 0 = 1 \\
 f(n + 1) = (n + 1) \times f n
 \end{array}
 \right.$$

4- Hilomorfismos Genéricos

Notação: $\llbracket g, h \rrbracket_F = \langle g \rangle_F \cdot \llbracket h \rrbracket_F$

Lei de fusão ou *deflorestação*, permite eliminar a construção de estruturas de dados intermédias:

$$\llbracket g, h \rrbracket_F = g \cdot F \llbracket g, h \rrbracket_F \cdot h$$

Ex: provar esta lei (general. do caso das listas)

XVI - Exemplos de Hilomorfismos

1- Algoritmo Mergesort

Tipo intermédio: LTree a

$$msort = ([\text{wrap}, \text{merge}]) \cdot [\text{separa}]$$

onde `wrap x = [x]`

e *merge* é a função de fusão de listas ordenadas.

Então o catamorfismo acima produz uma lista ordenada a partir de qualquer LTree

```
separa [x] = i1 x
separa l =
    let
        k = (length l) `div` 2
    in
        i2 (take k l, drop k l)
```

Então o anamorfismo com este gene constrói uma LTree equilibrada

Exercícios: (i) converter esta definição para código recursivo com variáveis
(ii) optimizar **separa**

Notas

- Dado o tipo de dados intermédio utilizado, esta função de ordenação não está definida para a lista vazia
- *Qualquer* árvore construída com os elementos da lista inicial serviria... esta escolha particular deve-se a questões de eficiência

2- Algoritmo Quicksort

Tipo intermédio: BTee a

$$qSort = ([[], \underline{combina}]) \cdot [separaPivot]$$

onde `combina (x, (l, r)) = l++(x:r)`

Então o catamorfismo acima é uma travessia *inorder* da árvore; produz uma lista ordenada a partir de uma BTee de pesquisa.

```
separaPivot [] = i1()
separaPivot (h:t) = i2 (h, aux h t)
  where aux x [] = ([],[])
        aux x (h:t)
          | h <= x = (h:a,b)
          | otherwise = (a,h:b)
  where (a,b) = aux x t
```

Então o anamorfismo com este gene constrói uma BTree de pesquisa, não necessariamente equilibrada

Exercício: converter esta definição para código recursivo com variáveis

3- Notas

- Depois de eliminada a estrutura de dados intermédia de um hilomorfismo, obtém-se uma função cuja **árvore de recursão** tem a forma dessa estrutura
- Nenhum dos exemplos apresentados corresponde a uma função que possa ser definida como anamorfismo ou catamorfismo (**porquê?**)
- Funções sobre listas com recursividade não trivial (ou seja, *não efectuada sobre a cauda da lista*) podem ser escritas como hilomorfismos de árvores
- Funções que constroem listas de forma não trivial (*não designando a cabeça e uma cauda construída recursivamente*) podem ser escritas como hilomorfismos de árvores

4- Arquitectura de Algoritmos

- *Ingredientes*: anamorfismos e catamorfismos standard, disponíveis em bibliotecas
- *Receitas simples*: composições diversas de catamorfismos com anamorfismos
- *Receitas sofisticadas*: composições ou outras combinações de hilomorfismos
- *Optimização*: simplificação dos algoritmos por deforestação.

5- Exemplo: Quicksort

$$qSort = ([[], \underline{combina}]) \cdot [separaPivot]$$

Alterando apenas uma das componentes deste hilo obtém-se facilmente outras funções úteis.

Por exemplo, utilizando o **anamorfismo** de *qSort* pode-se escrever uma função de pesquisa:

$$find\ x = ([\underline{False}, aux]) \cdot [separaPivot]$$

Como definir *aux*? Qual o tipo de *find*?

<code>aux (y, (l, r))</code>	<code> x==y = True</code>
	<code> x<y = l</code>
	<code> x>y = r</code>

Será esta uma função de pesquisa eficiente?
Em que situações fará sentido utilizá-la?

Outro exemplo: utilizando o **catamorfismo** de *qSort* escreve-se uma função de inversão de listas como:

$$rev = (\underline{[[]}}, combina) \cdot [separaInv]$$

Como definir *separaInv*?

```
separainv [] = i1()
separaInv l =
    let k = (length l) `div` 2
        l1 = take k l
        l2 = drop k l
    in i2(head l2, (tail l2, l1))
```

Será esta uma função de inversão eficiente?