CÁLCULO DE PROGRAMAS

P.R. Pereira

CONTEÚDO

PREÂMBULO

Este livro serve como suporte teórico para a unidade curricular de *Cálculo de Programas*[1] dos cursos LEI e LCC da Universidade do Minho. Tem como principal referência o livro *Program Design by Calculation*[2] do professor José N. Oliveira, regente da unidade curricular, e a sua estrutura segue a estrutura das aulas teóricas.

Universidade do Minho, Braga, Outubro 2021

Paulo R. Pereira

1

FUNÇÕES E TIPOS

Vamos voltar, por breves momentos, às aulas de matemática do secundário...

1.1 O QUE É UMA FUNÇÃO?

Uma função pode ser comparada a uma "caixa mágica" que recebe alguma coisa e produz alguma coisa.

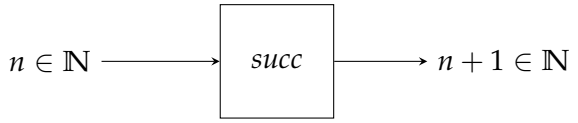
Suponhamos que temos um valor do tipo A e colocamos esse valor-A na "caixa mágica". Alguma coisa acontece e a caixa produz algo com esse valor-A.

O que é que a caixa produz? Depende do que a caixa faz. Por exemplo, se a caixa produzir o sucessor de um número natural e o dado valor-A é 10, a caixa irá produzir o valor 11. No mesmo exemplo, reparemos que a caixa só pode produzir um número natural! Se o dado valor-A é, por exemplo, a *string* "hello world", a caixa não saberá o que fazer, porque a caixa apenas produz sucessores de números naturais. É já intuitivo perceber que o tipo A corresponde ao conjunto dos números naturais.

Assim, neste caso, uma vez que o sucessor de um natural é também um natural, a "caixa" corresponde à seguinte função, a qual designamos por *succ* (*successor*):

$$succ : \mathbb{N} \to \mathbb{N}$$
$$succ\ n = n + 1$$

Podemos ver o seu processo no seguinte diagrama:



Um dos principais objetivos desta disciplina é o estudo da **programação composicional**, ou seja, aprender a escrever programas complexos por composição de programas mais simples (princípio da composicionalidade). Para tal, é necessário saber tipar corretamente uma função, isto é, identificar os conjuntos de valores aos quais pertencem os respetivos *inputs* (domínio da função) e *outputs* (conjunto de chegada da função). No exemplo dado em cima, facilmente identificamos estes dois conjuntos como o conjunto dos números naturais.

A isto chamamos a "assinatura" de uma função, e a notação que vamos usar para a descrever é bastante simples, correspondendo, neste caso, à seguinte:

$$\mathbb{N} \xleftarrow{succ} \mathbb{N}$$

O conjunto de chegada e o contradomínio da função nem sempre são o mesmo conjunto. Neste exemplo, como não existe um predecessor natural de 1, o conjunto de chegada difere do contradomínio, sendo este o conjunto $\mathbb{N} \setminus \{1\}$. No entanto, as funções devem ser assinadas com o seu conjunto de chegada e não com o contradomínio.

A função *succ* dá-nos uma visão muito matemática do que é uma função, mas é importante para introduzir a noção de cálculo na programação. Logicamente, os nossos programas não são funções assinadas por naturais, mas por estruturas complexas de dados. Contudo, os princípios da matemática aplicam-se (ou deveriam ser aplicados) de igual forma. Assim, o conjunto dos naturais corresponde ao **tipo de dados** de entrada e de saída da função *succ*.

O objetivo, a partir de agora, é trabalhar abstratamente com funções, olhando apenas para o seu tipo ou assinatura (o que recebem e o que retornam).

Assim, o tipo mais geral de uma função será

$$B \xleftarrow{\quad f \quad} A$$

onde a função f recebe um valor-A (o *input* pertence ao tipo de dados A) e produz um valor-B (o *output* pertence ao tipo de dados B).

> Os tipos de dados A e B são tipos arbitrários, podendo corresponder a qualquer tipo existente de dados.

Outras notações a serem usadas nesta disciplina para assinar a função f são $A \xrightarrow{f} B$, $f : B \leftarrow A$ e $f : A \to B$. Todas elas correspondem a escrever $f :: a \to b$ na linguagem de programação funcional HASKELL, onde as variáveis de tipo são escritas com letras minúsculas.

1.2 IGUALDADE FUNCIONAL

Formalmente, diremos que duas funções $f, g : B \leftarrow A$ são iguais se elas "concordarem" a nível *pointwise*, i.e.

$=_B$ denota igualdade ao nível do B.

$$f = g \quad sse \quad \langle \forall a : a \in A : f\ a =_B g\ a \rangle^1 \tag{1}$$

1.3 COMPOSIÇÃO FUNCIONAL

A composição funcional é o primeiro combinador funcional a ser estudado nesta disciplina. Portanto, sejam f e g as seguintes funções:

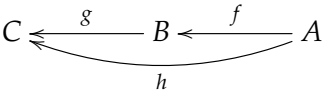
$$B \xleftarrow{\ f\ } A$$

$$C \xleftarrow{\ g\ } B$$

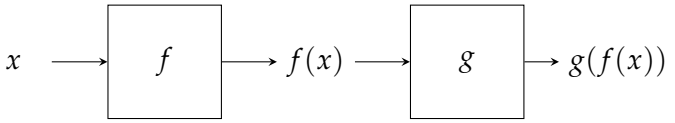
Vamos tentar "ligar" as duas setas. Obtemos, facilmente, o seguinte:

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

É já muito intuitiva a definição de uma outra função, h, que recebe um valor-A e produz um valor-C.



A função h corresponde à composição funcional de g com f. O conceito aprendido no secundário é o mesmo.



Analiticamente, a composição é definida da seguinte forma:

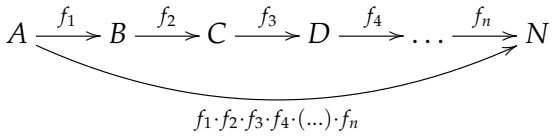
$$h\ x = (g \cdot f)\ x = g(f\ x) \tag{2}$$

1 $\langle \forall x : R : T \rangle$ significa: "para qualquer x em R, o termo T tem valor lógico verdadeiro", onde R e T são expressões lógicas que envolvem x.

e lemos "g após f".

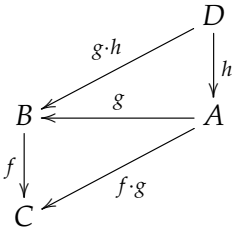
No nível pointfree, temos $h = (g \cdot f)$

A composição funcional é uma das bases desta disciplina. Podemos compor funcionalmente n funções, desde que o tipo de saída de cada função corresponda ao tipo de entrada da "função seguinte".



Sendo o mais básico de todos os combinadores funcionais, a maioria dos programadores nem sequer pensa nele quando querem combinar ou encadear funções para obter um programa mais elaborado. Isto deve-se a uma das suas mais importantes propriedades:

$$(f \cdot g) \cdot h = f \cdot (g \cdot h) \tag{3}$$



cujo nome é a propriedade *associativa* da composição.

Outra propriedade importante é a lei de *Leibniz*,

$$f \cdot h = g \cdot h \Leftarrow f = g \tag{4}$$

que nos diz que, dada a igualdade de duas funções, as composições das mesmas com uma outra função é também uma igualdade.

1.4 FUNÇÃO IDENTIDADE

A função identidade é aquela que, no contexto de uma função, é a menos útil. Não faz rigorosamente nada com o argumento, retornando-o tal como o recebeu. Contudo, será uma função muito importante, usada para preservar argumentos, como veremos mais tarde. É denominada de *id*, com a seguinte assinatura:

$$id : A \to A$$
$$id\ x \stackrel{\text{def}}{=} x$$

É já bastante intuitivo inferir a seguinte propriedade:

$$f \cdot id = id \cdot f = f \tag{5}$$

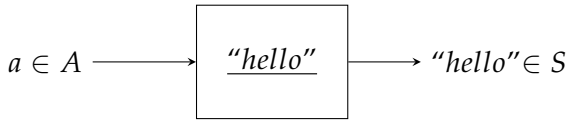
para uma qualquer função f. Esta propriedade é chamada de propriedade *natural* ou propriedade "grátis" da função *id*. Mais à frente, iremos descobrir porque lhe chamamos propriedade "grátis" e como inferir a propriedade natural de uma função polimórfica, isto é, uma função cujo tipo pode assumir várias formas ou padrões.

1.5 FUNÇÕES CONSTANTES

Uma função constante retorna sempre o mesmo valor, independentemente do argumento. Por exemplo, se g é a função constante que produz a string "*hello*", então,

```
g 23 = "hello"
g 'a' = "hello"
g [1,2,3,4] = "hello"
g "hello world" = "hello"
g "hello" = "hello"
```

A notação que vamos usar para descrever uma função constante é o sublinhado. No exemplo em cima, a função é definida em HASKELL como $g = const$ "*hello*", mas a notação que iremos usar é $g = \underline{\text{"hello"}}$.



Podemos expressar o tipo da função $\underline{\text{"hello"}}$ da seguinte forma:

S designa o tipo String.

$$S \xleftarrow{\underline{\text{"hello"}}} A$$

Para um qualquer $k \in K$, temos, por definição,

$$\underline{k}\ x \stackrel{\text{def}}{=} k \tag{6}$$

e, sendo f uma qualquer função,

$$\underline{k} \cdot f = \underline{k} \tag{7}$$
$$f \cdot \underline{k} = \underline{f\ k} \tag{8}$$

Estas duas últimas igualdades são conhecidas, respetivamente, como as propriedades *natural-constante* e *fusão-constante*.

Vamos agora introduzir o conceito de *preservação* de informação. A função identidade e as funções constantes são, por assim dizer, limites no "espectro funcional" referente à preservação dos dados.

A função identidade preserva todos os valores que recebe, enquanto que as funções constantes "deitam foram" todos os valores que recebem. Qualquer outra função está "entre" estas duas funções, perdendo "alguma" informação.

Como é que uma função perde informação? Basicamente de duas formas: ou "confunde" argumentos mapeando-os para o mesmo *output*, ou simplesmente "ignora" valores do seu conjunto de chegada.

1.6 MONOMORFISMOS

Um monomorfismo é uma função injetiva.

Uma função que "não confunde argumentos" é chamada de um *monomorfismo*. Como assim, "não confunde argumentos"?

Por exemplo, designemos g como a função que calcula o quadrado de um número real. Se o resultado de aplicar g a um determinado x for igual a 4, qual é o valor do x? Não sabemos! Poderá ser 2, mas também poderá ser -2.

Dizemos que g confunde os dois argumentos 2 e -2, uma vez que, embora sejam valores diferentes, a função produz o mesmo *output*.

Assim, uma função $B \xleftarrow{f} A$ diz-se um *monomorfismo* se, para qualquer par de funções $A \xleftarrow{h,k} C$, se $f \cdot h = f \cdot k$ então $h = k$, conforme nos mostra o seguinte diagrama:

$$B \xleftarrow{f} A \underset{k}{\overset{h}{\leftleftarrows}} C$$

Dizemos que f é "pós-cancelável".

1.7 EPIMORFISMOS

Um epimorfismo é uma função sobrejetiva.

Uma função que não "ignora valores" do seu conjunto de chegada é chamada de um *epimorfismo*. O que significa "ignorar valores" do conjunto de chegada?

Por exemplo, a função constante $g = \underline{10}$ têm como conjunto de chegada o conjunto dos números naturais, mas o único valor que a função produz é o 10. Todos os restantes naturais são ignorados.

Assim, uma função $A \xleftarrow{f} B$ diz-se um *epimorfismo* se, para qualquer par de funções $C \xleftarrow{h,k} A$, se $h \cdot f = k \cdot f$ então $h = k$, conforme nos mostra o seguinte diagrama:

$$C \underset{k}{\overset{h}{\leftleftarrows}} A \xleftarrow{f} B$$

Dizemos que f é "pré-cancelável".

1.8 ISOMORFISMOS

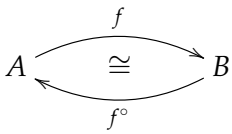
Um isomorfismo é uma função bijetiva.

Uma função que é, ao mesmo tempo, um monomorfismo e um epimorfismo é chamada de *isomorfismo* (ou *iso*).

A etimologia da palavra *isomorfismo* leva-nos ao grego antigo *isos* "igual" + *morphe* "forma", que nos permite definir informalmente um *isomorfismo* como uma função que mapeia valores de um tipo A para um tipo **equivalente** B, isto é, um tipo com "igual forma".

Assim, um *isomorfismo* $B \xleftarrow{f} A$ tem sempre uma função *inversa* $B \xrightarrow{f^\circ} A$.

Assim, e com a ajuda do seguinte diagrama,



podemos facilmente inferir as seguintes propriedades:

$$\begin{cases} f^\circ \cdot f &= id \\ f \cdot f^\circ &= id \end{cases} \tag{9}$$

Os isomorfismos são muito importantes pois podem converter dados de um *formato*, A, para um outro *formato*, B, sem perder informação. Estes formatos contêm a mesma informação, mas organizada de um modo diferente. Dizemos que A é isomórfico a B e escrevemos $A \cong B$ para expressar esse facto. Domínios de dados isomórficos são considerados *abstratamente* o mesmo.

Os isomorfismos são bastante flexíveis no cálculo ao nível *pointfree*. Se, por alguma razão, f° é mais fácil de desenvolver algebricamente do que f, então as seguintes regras são de grande ajuda:

$$f \cdot g = h \equiv g = f^\circ \cdot h \tag{10}$$

$$g \cdot f = h \equiv g = h \cdot f^\circ \tag{11}$$

Para compreender o que são *formatos* equivalentes, vamos considerar um exemplo. Relembremos a *propriedade distributiva* da álgebra elementar: $x\ (y + z) = x\ y + x\ z$.

Um número escrito no formato $x\ (y + z)$ pode ser reescrito no formato $x\ y + x\ z$, e vice-versa, sem sofrer qualquer alteração. As expressões $x\ (y + z)$ e $x\ y + x\ z$ são *formatos* equivalentes na representação de números.

Mais à frente iremos abordar alguns isomorfismos importantes. Mas, antes de o fazer, é necessário aprofundar o nosso conhecimento sobre combinadores funcionais. Em 1.3 revemos a composição de funções, o mais básico dos combinadores. Agora vamos introduzir novos combinadores funcionais: o Produto, o Coproduto e a Exponenciação.

1.9 PRODUTO E COPRODUTO

Como vimos em 1.3, no estudo da composição funcional, era necessário que o contradomínio de uma função estivesse contido no domínio da outra função para que estas pudessem ser compostas funcionalmente, matematicamente falando. Esta condição podia ser reescrita de uma forma mais simples na área das ciências da computação, com a introdução do conceito de tipos de dados. Portanto, vimos que a composição funcional de f com g exige que o tipo de saída de g corresponda ao tipo de entrada f.

Vamos estudar agora como combinar funções que "partilham" o mesmo domínio (o mesmo tipo de entrada) e como combinar funções que "partilham" o mesmo conjunto de chegada (o mesmo tipo de saída).

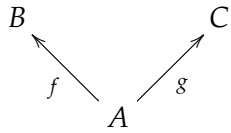
1.9.1 *Combinador Split*

Quando duas funções "partilham" o mesmo domínio, elas podem ser executadas em paralelo, originando um par. Por exemplo, sejam f e g as seguintes funções:

$$B \xleftarrow{\ f\ } A$$

$$C \xleftarrow{\ g\ } A$$

É possível "ligar" as duas setas e obter o seguinte diagrama:



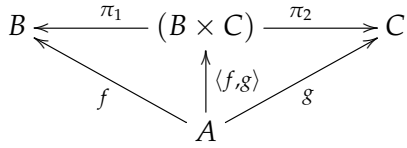
Agora, é mais intuitivo definir a função que combina f e g, produzindo um par que pertence ao produto cartesiano de B e C, isto é, ao seguinte conjunto:

$$B \times C = \{(b, c) \mid b \in B \ \wedge \ c \in C\}$$

Será usada a notação $\langle f, g \rangle$ para designar este combinador, "f *split* g", definido da seguinte forma:

$$\begin{array}{rcl} \langle f, g \rangle & : & A \to B \times C \\ \langle f, g \rangle \ a & \stackrel{\text{def}}{=} & (f \ a, \ g \ a) \end{array} \tag{12}$$

O seguinte diagrama expressa o seu tipo mais geral:



As funções π_1 e π_2 são as *projeções* do par (em HASKELL, $\pi_1 = fst$ e $\pi_2 = \ snd$), definidas da seguinte forma:

$$\pi_1(x, y) = x \qquad \pi_2(x, y) = y$$

O diagrama também oferece uma prova para a seguinte propriedade, conhecida como *cancelamento-*$\times$:

$$\pi_1 \cdot \langle f, g \rangle = f \quad \wedge \quad \pi_2 \cdot \langle f, g \rangle = g \tag{13}$$

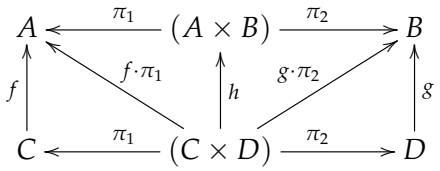
1.9.2 *Produto de Funções*

Quando duas funções não "partilham" o mesmo domínio não é possível utilizar o combinador *split*. No entanto, há algo que é possível fazer. Assumindo as seguintes funções:

$$A \xleftarrow{\ f\ } C$$

$$B \xleftarrow{\ g\ } D$$

vamos analisar o seguinte diagrama:



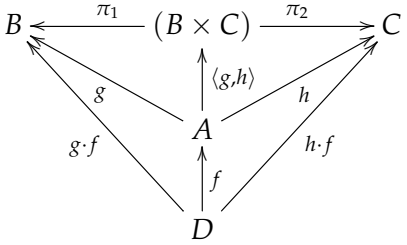
Podemos ver que $h = \langle f \cdot \pi_1, g \cdot \pi_2 \rangle$ está a mapear $C \times D$ para $A \times B$. Ora, isso corresponde à aplicação *paralela* de f e g e será expressa por $f \times g$. Assim, por definição:

$$f \times g \stackrel{\text{def}}{=} \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \tag{14}$$

1.9.3 *Propriedades do Produto*

- A composição funcional e o combinador *split* relacionam-se um com o outro através da propriedade *fusão-*$\times$:

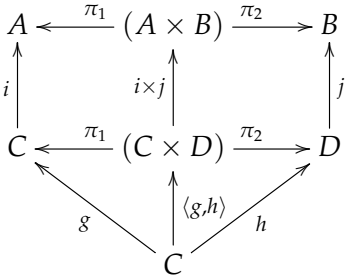
$$\langle g, h \rangle \cdot f = \langle g \cdot f, h \cdot f \rangle \tag{15}$$



- "O combinador *split* absorve o $\times$", como consequência da *fusão-*$\times$ e do *cancelamento-*$\times$[2] através da propriedade *absorção-*$\times$:

$$(i \times j) \cdot \langle g, h \rangle = \langle i \cdot g, j \cdot h \rangle \tag{16}$$

2 A propriedade *cancelamento-*$\times$ é explicada em 1.9.1



O diagrama também oferece duas outras propriedades sobre produtos e projeções:

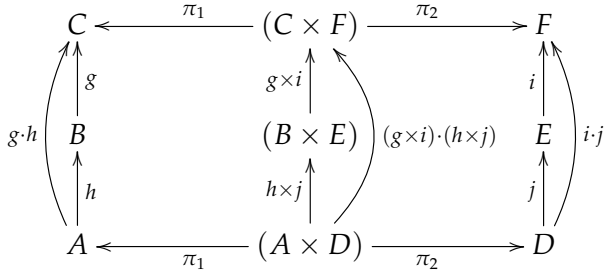
$$i \cdot \pi_1 = \pi_1 \cdot (i \times j) \tag{17}$$

$$j \cdot \pi_2 = \pi_2 \cdot (i \times j) \tag{18}$$

conhecidas como as propriedades *Natural-*π_1 e *Natural-*π_2, respetivamente.

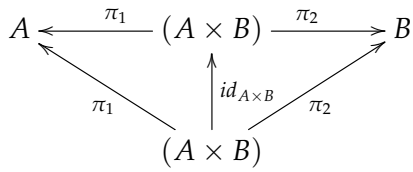
- Uma espécie de "bi-distribuição" do $\times$ com respeito à composição funcional através da propriedade *functor-*$\times$:

$$(g \cdot h) \times (i \cdot j) = (g \times i) \cdot (h \times j) \tag{19}$$



- O caso particular em que o combinador *split* é construido apenas à custa das projeções π_1 e π_2. Esta propriedade é conhecida como a *reflexão-*$\times$:

$$\langle \pi_1, \pi_2 \rangle = id_{A \times B} \tag{20}$$



Resumindo, as regras, com respeito aos tipos, dos combinadores *split* e *produto* são:

$$\frac{B \xleftarrow{f} A \qquad C \xleftarrow{g} A}{B \times C \xleftarrow{\langle f,g \rangle} A} \qquad \frac{C \xleftarrow{f} A \qquad D \xleftarrow{g} B}{C \times D \xleftarrow{f \times g} A \times B} \tag{21}$$

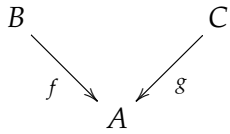
Vamo-nos focar agora em funções que "partilham" o mesmo contradomínio. Como é que as podemos "ligar"?

1.9.4 *Combinador Either*

Sejam f e g as seguintes funções:

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

Vamos tentar "ligar" as duas setas, tal como fizemos com o produto:



Este diagrama é muito semelhante ao diagrama do produto, diferindo apenas na direção das setas. No entanto, esta é uma grande diferença!

É fácil verificar que ambas as funções produzem um valor-A. Ou estamos no "lado B" e executamos f ou estamos no "lado C" e executamos g. Assim, podemos definir o combinador "either f or g" e a notação a ser usada é $[f, g]$. Obviamente, o seu conjunto de chegada será A. Mas e o que dizer do seu domínio? Talvez pensemos que seja $B \cup C$. Bem, isso funciona no caso em que B e C são conjuntos disjuntos, mas quando a interseção $B \cap C$ não é vazia, não sabemos se um determinado $x \in B \cap C$ veio do "lado B" ou do "lado C".

Assim, o domínio será a *união disjunta*, que corresponde ao conjunto[3]

$$B + C = \{i_1\ b \mid b \in B\}\ \cup\ \{i_2\ c \mid c \in C\}$$

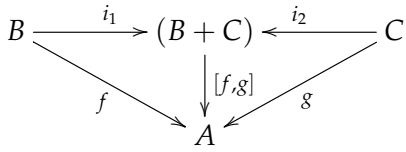
onde, em HASKELL, $i_1 = Left$ e $i_2 = Right$ cujos tipos são $B \xrightarrow{i_1} B + C \xleftarrow{i_2} C$ [4], e, no *Standard Prelude*, o tipo de dados $B + C$ é definido por:

```
data Either b c = Left b | Right c
```

Por fim, o combinador *either* é definido da seguinte forma:

$$\begin{array}{rcl} [f, g] & : & B + C \to A \\ [f, g]\ x & \stackrel{\text{def}}{=} & \begin{cases} (x = i_1\ b) \Rightarrow f\ b \\ (x = i_2\ c) \Rightarrow g\ c \end{cases} \end{array} \tag{22}$$

Como fizemos para o *produto*, podemos também definir o seguinte diagrama:



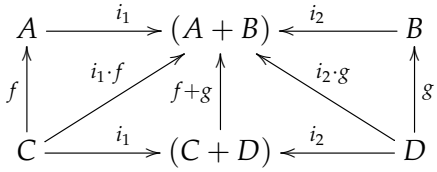
3 $B + C$ é chamado o *coproduto* de B e C.
4 As funções i_1 e i_2 são geralmente referidas como as *injeções* da união disjunta.

É interessante notar o quão semelhantes são os diagramas do Produto e do Coproduto – apenas invertemos as setas, substituímos as *projeções* por *injeções* e o *split* pelo *either*. Isto expressa o facto de que o produto e o coproduto são construções duais da matemática (tal como o seno e o cosseno da trigonometria). Esta dualidade traduz-se numa grande economia, uma vez que tudo o que dizer para o produto $A \times B$ pode ser transformado para o coproduto $A + B$, como veremos em 1.9.6.

1.9.5 *Coproduto de Funções*

Coproduto e soma são o mesmo

Tirando partido da referida dualidade, podemos introduzir a soma de funções $f + g$ como a notação dual do produto $f \times g$:



$$f + g \stackrel{\text{def}}{=} [i_1 \cdot f, i_2 \cdot g] \tag{23}$$

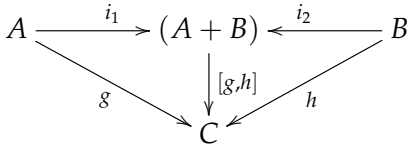
1.9.6 *Propriedades do Coproduto*

Tal como fizemos com o combinador Produto, podem ser definidas as seguintes propriedades:[5]

- *cancelamento-+*:

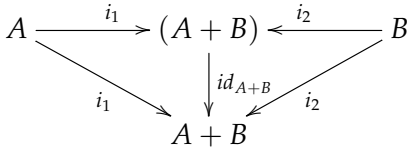
$$[g, h] \cdot i_1 = g, [g, h] \cdot i_2 = h \tag{24}$$

5 As propriedades são, no fundo, as mesmas do Produto, apenas transformadas para o combinador Coproduto, evidenciando assim a sua dualidade matemática.



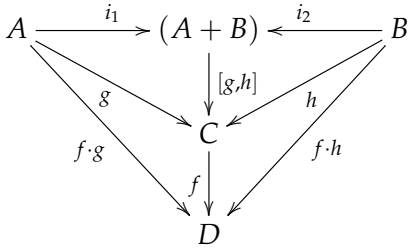
- *reflexão-+*:

$$[i_1, i_2] = id_{A+B} \tag{25}$$



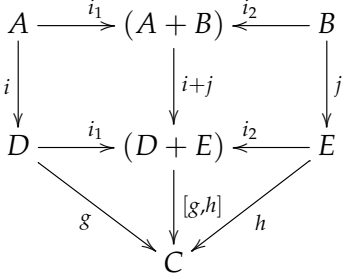
- *fusão-+*:

$$f \cdot [g, h] = [f \cdot g, f \cdot h] \tag{26}$$



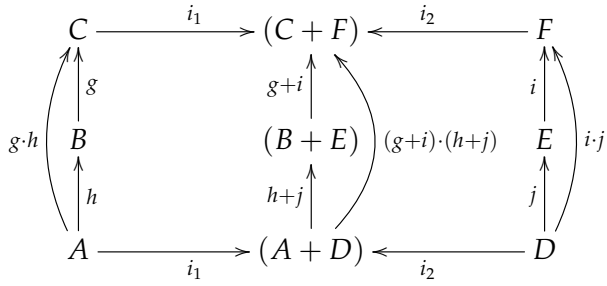
- *absorção-+*:

$$[g, h] \cdot (i + j) = [g \cdot i, h \cdot j] \tag{27}$$



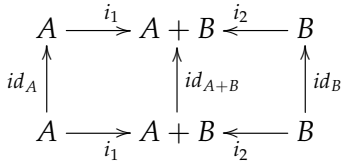
- *functor-+*:

$$(g \cdot h) + (i \cdot j) = (g + i) \cdot (h + j) \tag{28}$$



- *functor-id+*:

$$id_A + id_B = id_{A+B} \tag{29}$$



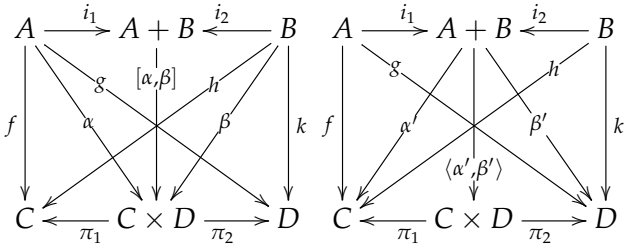
Resumindo, as regras, com respeito aos tipos, dos combinadores *either* e *coproduto* são:

$$\frac{C \xleftarrow{f} A \qquad C \xleftarrow{g} B}{C \xleftarrow{[f,g]} A + B} \qquad \frac{C \xleftarrow{f} A \qquad D \xleftarrow{g} B}{C + D \xleftarrow{f+g} A + B} \tag{30}$$

1.9.7 *Lei da Troca*

Uma função que mapeia valores de um coproduto $A + B$ para valores de um produto $A' \times B'$ pode ser expressa alternativamente como um *either* ou um *split*. Ambas as epxressões são equivalentes e esta equivalência traduz-se na *Lei da Troca*:

$$[\langle f, g\rangle, \langle h, k\rangle] = \langle [f, h], [g, k]\rangle \tag{31}$$



$$\sigma = [\alpha, \beta] \qquad \sigma' = \langle \alpha', \beta' \rangle$$
$$\alpha = \langle f, g \rangle \qquad \alpha' = [f, h] \qquad \sigma = \sigma'$$
$$\beta = \langle h, k \rangle \qquad \beta' = [g, k]$$

1.10 EXPONENCIAÇÃO

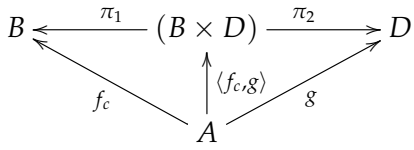
Suponhamos que queremos combinar duas funções com as seguintes assinaturas:

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

Nenhum dos combinadores já abordados, $f \cdot g$, $\langle f, g \rangle$ e $[f, g]$, pode ser usado diretamente. No entanto, o combinador $\langle f, g \rangle$ poderia ser usado se o componente C do domínio de f fosse "ignorado". Assim, suponhamos que, num contexto particular, o primeiro argumento de f é "irrelevante" e está restrito a um determinado $c \in C$. Deste modo, podemos derivar uma nova função a partir de f:

$$f_c : A \to B$$
$$f_c\ a \stackrel{\text{def}}{=} f(c, a)$$

Esta nova função combina com g através do combinador $\langle f_c, g \rangle$.



No que diz respeito à função f, existe uma grande diferença entre $f\ c$ e f_c.

$f\ c$ denota um valor do tipo B, *i.e.* $f\ c \in B$, mas, por outro lado, f_c denota uma função do tipo $B \leftarrow A$. Diremos que $f_c \in B^A$, introduzindo o novo tipo de dados, *exponencial*:

$$B^A \stackrel{\text{def}}{=} \{h \mid h : B \leftarrow A\} \tag{32}$$

Isto é, B^A corresponde ao conjunto de todas as funções cuja assinatura é $B \leftarrow A$.

Voltando à função $B \xleftarrow{f} C \times A$, podemos definir uma função f' que, para cada $c \in C$, produz $f_c \in B^A$. Ora, f' terá a seguinte assinatura:

$$B^A \xleftarrow{f'} C$$

Chamaremos a esta nova função f' a transposta de f e será usada a notação $\overline{f}$ para a designar.

Qual é a vantagem de $\overline{f}$ sobre f? A função $\overline{f}$ é mais "tolerante" do que f. Enquanto que f requer um par de argumentos ($c \in C$, $a \in A$) antes de poder ser aplicada, $\overline{f}$ "contenta-se" por receber primeiro um valor $c \in C$, e, depois, se a evolução do processo o requerer, um valor $a \in A$.

Como é que definimos funções transpostas em HASKELL? De forma muito simples. E, na verdade, não é algo novo para nós.

Como exemplo, suponhamos que queremos definir uma função que produz a soma de dois valores numéricos passados como argumentos. Existem duas formas de definir esta função:

$$soma\ (x,y) = x + y$$

ou

$$soma\ x\ y = x + y$$

A primeira versão da função *soma* recebe, claramente, um par. Sem o par, a função não pode ser aplicada. Necessita obrigatoriamente dos dois valores numéricos antes de ser aplicada. Contudo, o mesmo não acontece com a segunda versão. Qual é que é então a assinatura da segunda versão? Se questionarmos o GHCI com o tipo da função, obteremos o seguinte:

```
Prelude> soma x y = x + y
Prelude> :t soma
soma :: Num a => a -> a -> a
```

Portanto, a segunda versão da função *soma* recebe primeiro o valor x, e, depois, recebendo o valor y, produz a soma $x + y$. Assim sendo, a segunda versão da função *soma* tem a seguinte assinatura[6]:

$$\mathbb{N}^{\mathbb{N}} \xleftarrow{soma} \mathbb{N}$$

Questão: Olhando para o diagrama, é intuitivo pensar que deveria ser possível definir a função com apenas um argumento "do lado esquerdo". Como é que isso se procede?

A notação *lambda* é a resposta a essa pergunta.

$$f\ a = b \equiv f = \lambda a \to b \tag{33}$$

Assim, podemos definir a segunda versão da função *soma* do seguinte modo:

$$soma\ x = \lambda y \to x + y \tag{34}$$

6 Assumindo que a estamos a definir apenas para números naturais.

Esta notação permite-nos entender facilmente que esta versão recebe primeiro o x, e só depois, tendo o y, produz a soma $x + y$. Confirmemos, por fim, no *Standard Prelude* do HASKELL o que está a acontecer.

```
Prelude> soma x y = x + y
Prelude> :t soma
soma :: Num a => a -> a -> a
Prelude> soma x = \y -> x + y
Prelude> :t soma
soma :: Num a => a -> a -> a
```

Voltando, agora, às duas versões iniciais da função soma, é bastante intuitivo pensar que existem sempre duas formas de definir uma função que recebe dois argumentos, isto é, uma função f que recebe um valor do tipo A e um valor do tipo B e produz um valor do tipo C pode ser definida das duas seguintes formas:

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

$$C^B \xleftarrow{f'} A$$

É também muito razoável pensar que f é equivalente a f', alterando apenas na forma de construção. E portanto, uma vez que $f \in C^{A \times B}$ e $f' \in (C^B)^A$, também é razoável pensar que estes dois conjuntos são equivalentes, e de facto são. Esta equivalência é traduzida no seguinte isomorfismo:

$$C^{A \times B} \cong (C^B)^A$$

Existem, logicamente, funções que permitem mapear funções do tipo $C^{A \times B}$em$(C^B)^A$ e vice-versa. Essas funções estão definidas no *Standard Prelude* do HASKELL e são, respetivamente, as funções *curry* e *uncurry*. Mas, para prosseguir com o estudo deste isomorfismo, é necessário introduzir um novo operador, o operador *apply*, e as propriedades da exponenciação.

1.10.1 *Operador apply*

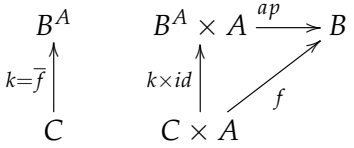
Como já referido, o tipo de dados exponencial B^A é habitado por funções que vão de A para B, e portanto, a declaração funcional $g : B \leftarrow A$ significa o mesmo que $g \in B^A$. O objetivo das funções é, naturalmente, servir de aplicação prática. E portanto, tendo um função f que transforma um valor do tipo A num valor do tipo B, e tendo um valor $a \in A$, podemos criar uma nova função que aplica f ao argumento a. Esta nova função tem o nome de operador *apply* e é definida do seguinte modo:

$$\begin{aligned} &ap : B \xleftarrow{ap} B^A \times A \\ &ap(f, a) \stackrel{\text{def}}{=} f\ a \end{aligned} \tag{35}$$

1.10.2 *Propriedades da Exponenciação*

Tal como o produto $A \times B$ e o coproduto $A + B$, o exponencial A^B também dispõe de uma propriedade universal:

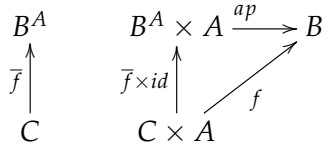
$$k = \overline{f} \equiv f = ap \cdot (k \times id) \tag{36}$$



Da propriedade universal, são derivadas as seguintes propriedades:

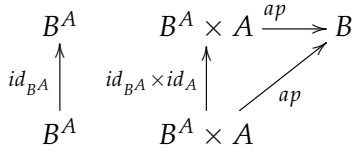
- *cancelamento-exp*:

$$f = ap \cdot (\overline{f} \times id) \tag{37}$$



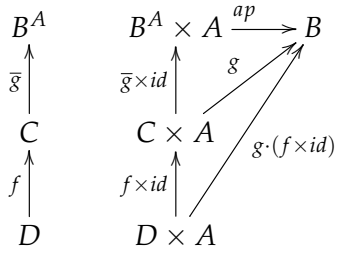
- *reflexão-exp*:

$$\overline{ap} = id_{B^A} \tag{38}$$



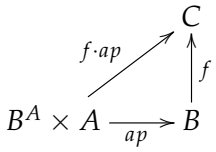
- *fusão-exp*:

$$\overline{g \cdot (f \times id)} = \overline{g} \cdot f \tag{39}$$

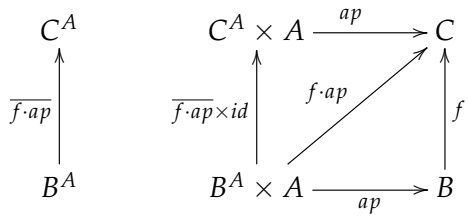


Para introduzir as propriedades seguintes, é necessário introduzir um novo combinador funcional que nasce da transposta de $f \cdot ap$. Qual é então a assinatura de $\overline{f \cdot ap}$?

Comecemos por perceber o que significa $f \cdot ap$.



Pelo conhecimento que já temos sobre o combinador exponencial, já conseguirmos inferir que a transposta de $f \cdot ap$ terá a assinatura $\overline{f \cdot ap} \ : \ B^A \to C^A$.



A notação que vamos usar para designar $\overline{f \cdot ap}$ será f^A e aplica-se a seguinte regra:

$$\frac{C \xleftarrow{f} B}{C^A \xleftarrow{f^A} B^A}$$

O que é que f^A significa na prática?

f^A recebe como argumento uma função $g : A \to B$. Portanto, fixando uma função $f : B \to C$ e um tipo de dados A, tendo um determinado $a \in A$, a função argumento g produz, com esse a, um $g\ a = b \in B$ e "passa" o resultado a f, que aceita o b e produz um $c \in C$. Ora, isso é precisamente a composição funcional já abordada anteriormente. Portanto, f^A é o combinador funcional "composição com f":

$$(f^A)g \stackrel{\text{def}}{=} f \cdot g \tag{40}$$

De facto,

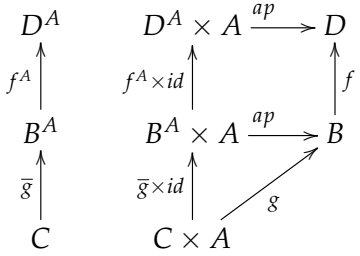
$$
\begin{aligned}
& \overline{f \cdot ap} = f^A \\
\equiv \quad & \{ \text{ universal-exp } \} \\
& ap \cdot (f^A \times id) = f \cdot ap \\
\equiv \quad & \{ \text{ igualdade extensional } \} \\
& (ap \cdot (f^A \times id))(g, a) = (f \cdot ap)(g, a) \\
\equiv \quad & \{ \text{ def-comp; def-}\times\text{; def-id } \} \\
& ap(f^A\ g, a) = f(ap(g, a)) \\
\equiv \quad & \{ \text{ def-}\textit{apply} \text{ } \} \\
& (f^A\ g)\ a = f(g\ a) \\
\equiv \quad & \{ \text{ def-comp } \} \\
& (f^A\ g)\ a = (f \cdot g)\ a \\
\equiv \quad & \{ \text{ igualdade extensional } \} \\
& f^A\ g = f \cdot g
\end{aligned}
$$

□

Estamos agora prontos para entender as seguintes propriedades:

- *absorção-exp*:

$$\overline{f \cdot g} = f^A \cdot \overline{g} = \overline{f \cdot ap} \cdot \overline{g} \tag{41}$$



Note, no mesmo diagrama, como também inferimos que

$$f^A = \overline{f \cdot ap} \tag{42}$$

- *functor-exp*:

$$(g \cdot h)^A = G^A \cdot h^A \tag{43}$$

- *functor-id-exp*:

$$id^A = id \tag{44}$$

1.10.3 *Curry e Uncurry*

Conforme já mencionado, as funções *curry* e *uncurry*, definidas no *Standard Prelude* do HASKELL, permitem-nos mapear funções do tipo $C^{A\times B}$ em $(C^B)^A$ e vice-versa, respetivamente.

$$C^{A\times B} \underset{uncurry}{\overset{curry}{\cong}} (C^B)^A \tag{45}$$

```
curry :: ((a, b) → c) → a → b → c
curry f a b = f (a, b)

uncurry :: (a → b → c) → (a, b) → c
uncurry f (a, b) = f a b
```

Dizemos que as funções do tipo $C^{A\times B}$ estão "uncurried" e as funções do tipo $(C^B)^A$ estão "curried".

Vamos agora calcular o significado de *curry* por remover variáveis da sua definição:

$$
\begin{aligned}
& \overbrace{\underbrace{\text{curry } f}_{\overline{f}}\ a}^{g}\ b = f(a,b) \\
\equiv & \quad \{ \text{ introdução de } g \ \} \\
& g\ b = f(a,b) \\
\equiv & \quad \{ \text{ def-}\textit{apply} \ \} \\
& ap(g,b) = f(a,b) \\
\equiv & \quad \{ \ g = \overline{f}a; \text{ def-id } \} \\
& ap(\overline{f}\ a, id\ b) = f(a,b) \\
\equiv & \quad \{ \text{ def-}\times \ \} \\
& ap((\overline{f} \times id)(a,b)) = f(a,b) \\
\equiv & \quad \{ \text{ def-comp; igualdade extensional } \} \\
& ap \cdot (\overline{f} \times id) = f
\end{aligned}
$$

Inferimos assim que a definição de *curry* é uma reformulação da lei de cancelamento. Isto é,

$$\text{curry } f \stackrel{\text{def}}{=} \overline{f} \tag{46}$$

e corresponde à transposta na linguagem HASKELL.

Vamos fazer o mesmo com a função *uncurry*:

$$
\begin{aligned}
& \underbrace{\text{uncurry } f}_{k}(a,b) = f\ a\ b \\
\equiv & \quad \{ \text{ introdução de } k\text{; cancelamento-exp } \} \\
& k\ (a,b) = (ap \cdot (f \times id))\ (a,b) \\
\equiv & \quad \{ \text{ igualdade extensional } \} \\
& k = ap \cdot (f \times id) \\
\equiv & \quad \{ \text{ universal-exp } \} \\
& f = \overline{k} \\
\equiv & \quad \{ \text{ definição de } k \} \\
& f = \overline{\text{uncurry } f}
\end{aligned}
$$

Portanto, *uncurry* é a inversa da transposta, *i.e.*, inversa de *curry*. A notação a ser usada para designar *uncurry* f será $\widehat{f}$, de modo que o isomorfismo apresentado em 45 pode ser traduzida por $f = \overline{\widehat{f}}$. Naturalmente, $f = \widehat{\overline{f}}$ também se verifica, instanciando k por $\widehat{\overline{f}}$, na equação em cima.

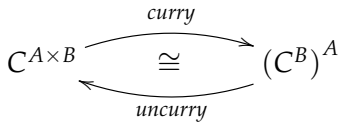
$$
\begin{aligned}
& \widehat{\overline{f}} = ap \cdot (\overline{f} \times id) \\
\equiv & \quad \{ \text{ cancelamento-exp } \} \\
& \widehat{\overline{f}} = f
\end{aligned}
\qquad \square
$$

Portanto,

$$g = \overline{f} \Leftrightarrow \widehat{g} = f \tag{47}$$

levando ao isomorfismo já apresentado

$$A \to C^B \ \cong \ A \times B \to C$$



1.11 FUNCTORES

O conceito de functor é nos emprestado da Teoria das Categorias e é muito útil na álgebra da programação.

Um functor **F** pode ser considerado como um construtor de tipos de dados que, dado um tipo A, constrói um tipo de dados mais elaborado **F** A. Também, dado um outro tipo B, constrói, de forma semelhante, um tipo de dados mais elaborado **F** B, e assim por diante.

O mais relevante é que a estrutura criada pelo functor é também estendida a funções. Como assim?

Dada a função $B \xleftarrow{f} A$, reparemos que A e B são parâmetros de **F** A e **F** B, respetivamente. Assim, sendo f uma função que transforma valores do tipo A em valores do tipo B, podemos definir a função $\mathbf{F}\ B \xleftarrow{\mathbf{F}\ f} \mathbf{F}\ A$ que, para cada valor do tipo A da estrutura **F** A, aplica a função f, dando como resultado uma estrutura **F** B.

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

Dizemos que **F** f estende f para estruturas-**F**, e, por definição, obedece às seguintes propriedades:

- Functor-F:

$$\mathbf{F}\ (g \cdot h) = (\mathbf{F}\ g) \cdot (\mathbf{F}\ h) \tag{48}$$

- Functor-id-F:

$$\mathbf{F}\ id_A = id_{(\mathbf{F}\ A)} \tag{49}$$

Os dois functores básicos são:

- Functor identidade:

$$\begin{array}{ccc} A & \cdots & A \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f} \\ B & \cdots & B \end{array} \qquad \begin{array}{l} \mathbf{F}\ X = X \\ \mathbf{F}\ f = f \end{array} \tag{50}$$

- Functor constante:

$$\begin{array}{ccc} A & \cdots & C \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id_C} \\ B & \cdots & C \end{array} \qquad \begin{array}{l} \mathbf{F}\ X = C \\ \mathbf{F}\ f = id_C \end{array} \tag{51}$$

1.11.1 *Functores Polinomiais*

Functores polinomiais são descritos por expressões polinomiais, como por exemplo:

$$\mathbf{F}\ X = 1 + A \times X$$

Assim, um functor polinomial pode ser um dos seguintes três casos:

- o functor identidade ou um functor constante;
- o produto ou coproduto (soma) finito de functores polinomiais;
- a composição de functores polinomiais.

Dado o exemplo em cima, podemos derivar o seguinte:

$$
\begin{aligned}
\mathbf{F}\ f &= (1 + A \times X)f \\
&= \quad \{ \text{ soma de functores } \} \\
&\quad (1)f + (A \times X)f \\
&= \quad \{ \text{ functor constante e produto de functores } \} \\
&\quad id_1 + (A)f \times (X)f \\
&= \quad \{ \text{ functor constante e functor identidade } \} \\
&\quad id_1 + id_A \times f \\
&= \quad \{ \text{ simplificação } \} \\
&\quad id + id \times f
\end{aligned}
$$

Observemos que $1 + A \times X$ denota o mesmo que $id + id \times f$. O que é que isso significa na prática? Ao integrar o tipo de dados X no tipo $1 + A \times X$, a função usada para percorrer a nova estrutura será $id + id \times f$, assumindo que f é uma função que opera sobre X e que não queremos alterar os restantes elementos do tipo. Assim, podemos pensar num functor como um "par de funções". Ainda no exemplo dado, obtemos duas funções:

$$
\begin{cases}
\mathbf{F}\ X = 1 + A \times X \\
\mathbf{F}\ f = id + id \times f
\end{cases}
$$

$\mathbf{F}\ X$ integra o tipo de dados X num novo tipo e $\mathbf{F}\ f$ é função usada para trabalhar sobre o novo tipo de dados.

O facto de a expressão polinomial usada para os dados ser a mesma expressão para os operadores que transformam estruturalmente tais dados é de grande aplicação prática, pois permite-nos definir os seguintes diagramas:

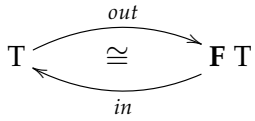
$$
\begin{array}{ccc}
\mathrm{T} & \xrightarrow{out_{\mathrm{T}}} & \mathbf{F}\ \mathrm{T} \\
{\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathbf{F}\ f} \\
B & \xleftarrow[g]{} & \mathbf{F}\ B
\end{array}
\qquad\qquad
\begin{array}{ccc}
\mathrm{T} & \xleftarrow{in_{\mathrm{T}}} & \mathbf{F}\ \mathrm{T} \\
{\scriptstyle h}\uparrow & & \uparrow{\scriptstyle \mathbf{F}\ h} \\
B & \xrightarrow[k]{} & \mathbf{F}\ B
\end{array}
$$

onde a função *out* "decompõe" o tipo de dados T numa determinada estrutura e a função *in* "constrói", a partir dessa mesma estrutura, o tipo de dados em causa.

Na Teoria das Categorias, diz-se que a função *in* pertence a uma classe própria de funções chamadas de *F-algebras*.

Uma *F-algebra* é qualquer função α com a assinatura $A \xleftarrow{\alpha} \mathbf{F}\ A$ e pode ser monomórfica, epimórfica ou isomórfica.

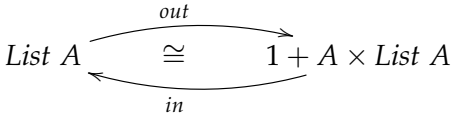
Quando estamos perante uma *F-algebra* isomórfica, é imediatamente intuitivo o seguinte diagrama:



que será extremamente importante na introdução da recursividade ao nível *pointfree* no capítulo 3.

Designamos T como um *tipo indutivo* e **F** T como o *functor tipo* associado a T.

Por exemplo, pensemos no tipo de dados *List* (listas ou sequências finitas). Um lista ou é vazia ou é composta por uma cabeça (primeiro elemento da lista) e uma cauda (a restante lista).[7] Assim, podemos definir o seguinte *isomorfismo* de construção de listas:



Se a lista for vazia, a função *out* irá injetar o elemento vazio à esquerda, caso contrário irá injetar o par (cabeça da lista, cauda da lista) à direita.

7 No caso da lista singular, a cauda é a lista vazia.

O tipo indutivo em causa é *List A*, e o seu functor tipo associado é $1 + A \times List\ A$.

O tipo 1 é unicamente habitado pelo elemento vazio. Em HASKELL, o elemento vazio corresponde ao $()$. Então, se estivermos perante o valor $i_1()$, sabemos que a lista original corresponde à lista vazia. Por outro lado, se estivermos, por exemplo, perante o valor $i_2(1, [2, 3])$, a lista original corresponde à lista $[1, 2, 3]$.

A notação a ser usada para listas de valores de um qualquer tipo A será A^*. Assim sendo, podemos redesenhar o diagrama da seguinte forma:

$$A^* \underset{in}{\overset{out}{\cong}} 1 + A \times A^*$$

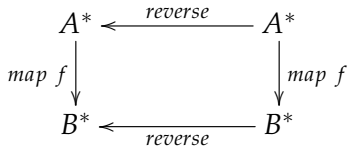
1.11.2 *Propriedades Naturais*

A propriedade natural é geralmente chamada de *propriedade grátis*, uma vez que é facilmente derivada de qualquer função polimórfica, isto é, cujo domínio e conjunto de chegada podem assumir várias "formas" ou "padrões".

$$\begin{array}{ccc} A & & \mathbf{F}\ A \xleftarrow{\phi} \mathbf{G}\ A \\ {\scriptstyle f}\downarrow & & {\scriptstyle \mathbf{F}\ f}\downarrow \qquad\qquad \downarrow{\scriptstyle \mathbf{G}\ f} \\ B & & \mathbf{F}\ B \xleftarrow[\phi]{} \mathbf{G}\ B \end{array} \qquad (\mathbf{F}\ f) \cdot \phi = \phi \cdot (\mathbf{G}\ f) \qquad (52)$$

f é quantificada universalmente, ou seja, a propriedade *natural* é válida para qualquer $f : B \leftarrow A$.

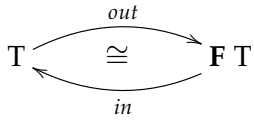
Exemplo: a propriedade natural da função *reverse* (função que inverte uma lista) é a seguinte:



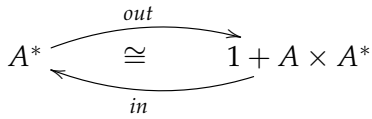
$$map\ f \cdot reverse = reverse \cdot map\ f$$

1.12 BI-FUNCTORES

Em 1.11 introduzimos a noção de *tipo indutivo* e do seu *functor tipo* associado, conforme representado no seguinte diagrama:



Vimos também o caso particular das listas ou sequências finitas.



Mas, existe um problema!

Vejamos este caso em particular, onde T corresponde a A^*. Como é que definimos **F** T ?

$$\mathbf{F}\ \mathrm{T} = 1 + ? \times \mathrm{T}$$

Não conseguimos definir, pois T não nos permite ter uma visão paramétrica de si mesmo, isto é, não sabemos qual é o tipo de dados que compõe a lista. Surge então a necessidade de um functor binário que nos permita definir **F** T.

Supondo que apenas um parâmetro é identificado em T, definimos

$$\mathrm{T}\ X = \mathbf{B}(X, \mathrm{T}\ X)$$

onde $\mathbf{B}$ é o *bi-functor de base*[8] do tipo T.

No exemplo dado em cima,

$$\mathbf{B}(X, Y) = 1 + X \times Y$$

e, como A^* (ou *List* A) corresponde ao T X,

$$A^* \cong \mathbf{B}(A,\ A^*)$$
$$A^* \cong 1 + A \times A^*$$

1.13 ALGUNS TIPOS INDUTIVOS

Designação	T	$\mathbf{B}(X, Y)$	$\mathbf{B}(f, g)$
Naturais	$\mathbb{N}_0$	Não existe	Não existe
Sequências (listas finitas)	A^*	$1 + X \times Y$	$id + f \times g$
Sequências não vazias	A^+	$X + X \times Y$	$id + f \times g$
Leaf Trees	LTree A	$X + Y^2$	$f + g^2$
Binary Trees	BTree A	$1 + X \times Y^2$	$id + f \times g^2$

O tipo de dados $\mathbb{N}_0$ não pode ser parametrizado e por isso não é possível definir $\mathbf{B}(X, Y)$ e $\mathbf{B}(f, g)$. Neste caso, $\mathbf{F}\ X = 1 + X$.
Os naturais já são em si um tipo concreto de dados. Por exemplo, listas e árvores são necessariamente paramétricas. Listas de quê? Árvores de quê? De *booleanos*, de inteiros, de *strings*, etc. Não existem naturais de... Existem naturais.

8 O seu nome deve-se ao facto de ser a base de toda a definição de tipo indutivo.

2

CONDICIONAL DE MCCARTHY

O condicional de McCarthy permite-nos escrever *if clauses* ao nível *pointfree*. Relembremos o tradicional "if then else":

$$y = \text{if } p(x) \text{ then } f(x) \text{ else } g(x)$$

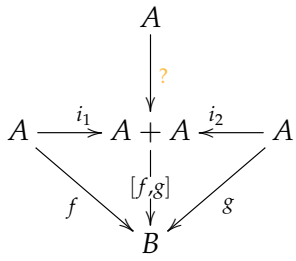
onde p é um predicado envolvendo x.

Se $p(x)$ for verdadeiro, executamos $f(x)$, caso contrário, executamos $g(x)$. Uma coisa é certa: ou f é executada ou g é executada. Ora, nós sabemos escrever isto no nível *pointfree*, basta recorrer ao combinador *either*, isto é, $[f, g]$.

Sendo x um valor de um tipo qualquer A, ambas as funções, f e g, vão ter como domínio o tipo A, e, com esse valor x, vão produzir um dado y de um tipo qualquer B.[1]

1 A e B são tipos arbitrários que representam qualquer tipo de dados. É perfeitamente possível que haja casos em que A e B sejam exatamente o mesmo tipo de dados.

Podemos assim definir um diagrama que expressa tudo o que já foi considerado.



Falta apenas definir o que fazer antes de executar $[f, g]$. Correr f ou g dependerá do resultado de aplicar o predicado p. Se $p(x)$ for verdadeiro, injetamos x no lado esquerdo de $A + A$. Caso contrário, injetamos do lado direito.

Para definir este passo ao nível *pointfree*, é necessário introduzir os seguintes isomorfismos:

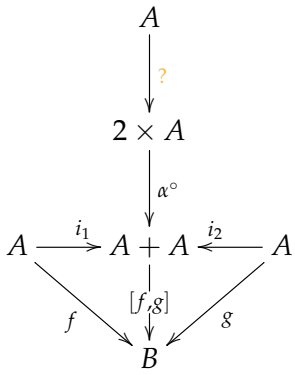
$$2 \cong \mathbb{B}$$

$$2 \times A \underset{\alpha}{\overset{\alpha^\circ}{\cong}} A + A$$

$$\alpha = [\langle \underline{True}, id \rangle, \langle \underline{False}, id \rangle]$$

$$\alpha^\circ\ (True, x) = i_1\ x$$
$$\alpha^\circ\ (False, x) = i_2\ x$$

A seguinte evolução no diagrama é agora intuitiva:



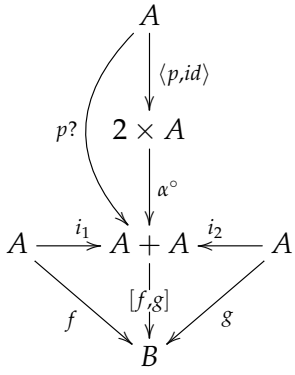
Falta apenas correr o predicado $p(x)$, preservando, paralelamente, o argumento. Isto é, falta apenas definir a função $2 \times A \xleftarrow{?} A$.

Por um lado, executa-se o predicado p dando origem a um *booleano* ($2 \cong \mathbb{B}$), e por outro, preserva-se o argumento com recurso à função identidade.

Obtém-se:

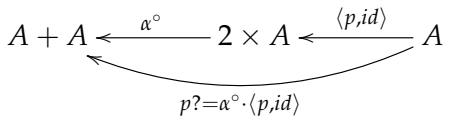
$$2 \times A \xleftarrow{\langle p,id \rangle} A$$

finalizando o diagrama e introduzindo a noção de *guarda*:



Designamos $p?$ como o *guarda* associado ao predicado p. O guarda do predicado é muito mais informativo do que o predi-

cado em si, uma vez que que já nos dá o resultado de testar p a um dado *input*.



A notação *pointfree* a ser usada é conhecida como o "Condicional de McCarthy"[2] e é muito intuitiva:

$$p \to f, g \stackrel{\text{def}}{=} [f, g] \cdot p?$$

É pronunciado da seguinte forma: "se p então f, senão g".

2.1 PROPRIEDADES

- Natural-guarda:

$$p? \cdot f = (f + f) \cdot (p \cdot f)? \tag{53}$$

- 1ª Lei de fusão:

$$f \cdot (p \to g, h) = p \to f \cdot g, f \cdot h \tag{54}$$

- 2ª Lei de fusão:

$$(p \to f, g) \cdot h = (p \cdot h) \to (f \cdot h), (g \cdot h) \tag{55}$$

2 Depois do cientista da computação John McCarthy o definir.

3

CATAMORFISMOS

Catamorfismo é uma palavra que deriva do grego antigo: *cata* "para baixo" + *morphe* "forma". Assim, a etimologia da palavra *catamorfismo* permite-nos chegar, de forma intuitiva, a uma definição informal de *camorfismo* – uma função que "consome" (ou "destrói") uma determinada estrutura. Vejamos um exemplo.

3.1 CATARMOFISMO SOBRE LISTAS

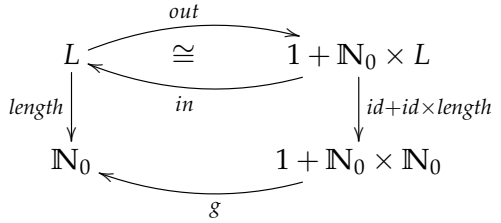
Vamos relembrar o *isomorfismo* de construção de listas estudado em 1.12. Assim, seja L o tipo que representa listas de naturais. Estamos então perante o seguinte isomorfismo:

$$L \underset{in}{\overset{out}{\cong}} 1 + \mathbb{N}_0 \times L$$

Pensemos agora no algoritmo para calcular o comprimento de uma lista – a função *length*, definida em HASKELL do seguinte modo:

```
length [] = 0
length (h:t) = 1 + length t
```

e vamos tentar completar o seguinte diagrama, definindo a função g:



Notemos que g "sai" de uma soma e portanto usar-se-á o combinador *either*. Vamos começar então por definir $g = [g_1, g_2]$.

Ao executar o *out*, o "caso da esquerda" é a lista vazia, cujo elemento $()$ é preservado pela função *id*. Assim, a primeira função do *either*, g_1, tratará do caso em que a lista é vazia e recebe o valor $()$. Ora, o comprimento de uma lista vazia é o e portanto $g_1() = 0$, que pode ser substituída pela função constante $\underline{0}$. Assim, a nossa função g é agora definida como:

$$g = [\underline{0}, g_2]$$

O que dizer de g_2? A função g_2 trata do caso em que a lista não é vazia. Notemos que o elemento da cabeça da lista é preservado pela função identidade, mas, é aplicada a função *length* à sua cauda. Assim, a função g_2 recebe um par (x, y) no qual x é o primeiro elemento da lista e y o comprimento da cauda. Ora, o comprimento de uma lista não vazia é o sucessor do comprimento da cauda. Assim, $g_2(x, y) = succ\ y$, que pode ser reescrita no nível *pointfree* como $g_2 = succ \cdot \pi_2$.

Por fim, temos:

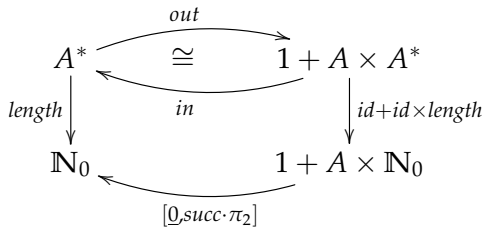
$$g = [\underline{0}, succ \cdot \pi_2]$$

Diremos que g é o *gene* do catamorfismo *length* e usar-se-ão os *banana brackets* como notação. Assim,

$$length = (\!|\, [\underline{0}, succ \cdot \pi_2] \,|\!)$$

Neste exemplo, o tipo L corresponde ao tipo ${\mathbb{N}_0}^*$, isto é, listas de naturais.

Podemos tornar a função *length* polimórfica, pois o comprimento de uma lista não depende do tipo de dados que a compõem. Assim, obtemos o seguinte diagrama:



$$
\begin{aligned}
& length \cdot in = [\underline{0}, succ \cdot \pi_2] \cdot (id + id \times length) \\
\equiv & \quad \{ \text{ definição de } in \text{; absorção-+; natural-id } \} \\
& length \cdot [nil, cons] = [\underline{0}, (succ \cdot \pi_2) \cdot (id \times length)] \\
\equiv & \quad \{ \text{ fusão-+; eq-+ } \} \\
& \begin{cases} length \cdot nil = \underline{0} \\ length \cdot cons = (succ \cdot \pi_2) \cdot (id \times length) \end{cases} \\
\equiv & \quad \{ \text{ igualdade extensional; def-comp; def-const; def-}x \text{ } \} \\
& \begin{cases} length\ (nil\ x) = 0 \\ length\ (cons\ (h, t)) = succ\ (\pi_2(id\ h, length\ t)) \end{cases} \\
\equiv & \quad \{ \text{ def-nil; def-cons; def-id; def-proj; definição de } succ \} \\
& \begin{cases} length\ [\,] = 0 \\ length\ (h : t) = (length\ t) + 1 \end{cases}
\end{aligned}
$$

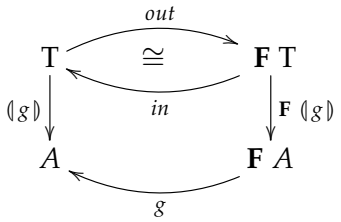
Obtivemos exatamente a mesma definição da função *length* apresentada em HASKELL no início da secção.

A última questão é: reconhecemos $(id + id \times length)$ e $(1 + A \times A^*)$?

Sim! Estamos perante o functor do tipo lista de valores-A. O que seria de esperar, visto se tratar de um catamorfismo sobre listas. Assim, é mais intuitivo generalizar o conceito de catamorfismo, para um qualquer tipo indutivo T.

3.2 GENERALIZAÇÃO

Diagrama geral de um catamorfismo $(\!|\, g \,|\!)$ para um tipo indutivo T:



Podemos também inferir algumas propriedades a partir deste diagrama.

3.3 PROPRIEDADES

- *universal-cata*:

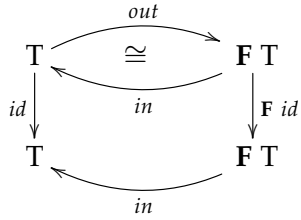
$$k = (\!|\, g \,|\!) \equiv k \cdot in = g \cdot \mathbf{F}\, k \tag{56}$$

- *cancelamento-cata*:

$$(\!|\, g \,|\!) \cdot in = g \cdot \mathbf{F}\, (\!|\, g \,|\!) \tag{57}$$

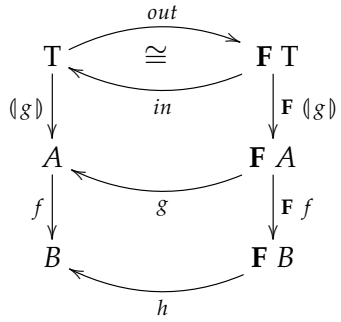
- *reflexão-cata*:

$$(\!|\, in \,|\!) = id_{\mathrm{T}} \tag{58}$$



- *fusão-cata*:

$$f \cdot (\!|\, g \,|\!) = (\!|\, h \,|\!) \Leftarrow f \cdot g = h \cdot \mathbf{F}\, f \tag{59}$$



- *absorção-cata*:

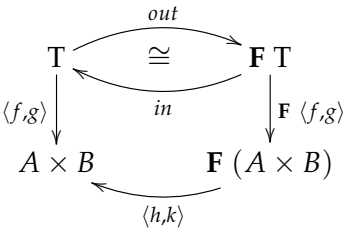
$$(\!|\, g \,|\!) \cdot \mathrm{T}\, f = (\!|\, g \cdot \mathbf{B}\, (f, id) \,|\!) \tag{60}$$

4

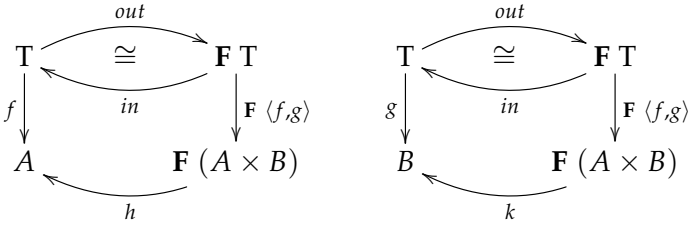
RECURSIVIDADE MÚTUA

O que dizer sobre catamorfismos que geram pares como resultado? Bem, o seu gene terá de ser, obviamente, um *split*. No entanto, isso, por si só, não nos traz nada de novo. Qual é então a grande vantagem que estes catamorfismos nos trazem?

Vejamos o seguinte diagrama:



Do diagrama, inferimos que $\langle f, g \rangle = (\!| \langle h, k \rangle |\!)$, e, com a ajuda dos seguintes diagramas,



chegamos facilmente à seguinte conclusão:

$$
\begin{aligned}
& \langle f, g \rangle = (\!| \langle h, k \rangle |\!) \\
\equiv & \quad \{ \text{ universal-cata } \} \\
& \langle f, g \rangle \cdot in = \langle h, k \rangle \cdot \mathbf{F}\ \langle f, g \rangle \\
\equiv & \quad \{ \text{ fusão-}\times \text{ 2x } \} \\
& \langle f \cdot in, g \cdot in \rangle = \langle h \cdot \mathbf{F}\ \langle f, g \rangle, k \cdot \mathbf{F}\ \langle f, g \rangle \rangle
\end{aligned}
$$

É já bastante intuitivo perceber que f e g, além de serem executadas em paralelo, partilham informação iterativamente, i.e. são mutuamente recursivas.

Assim, recorrendo à propriedade *Eq*-$\times$, introduzimos a Lei de Recursividade Mútua:

$$
\begin{cases} f \cdot in = h \cdot \mathbf{F}\ \langle f, g \rangle \\ g \cdot in = k \cdot \mathbf{F}\ \langle f, g \rangle \end{cases} \equiv \langle f, g \rangle = (\!| \langle h, k \rangle |\!) \tag{61}
$$

Esta lei, também conhecida como a "lei de Fokkinga", permite combinar duas funções mutuamente recursivas num único catamorfismo.

Pode-se, inclusive, aplicar a lei de recursividade mútua a mais do que duas funções mutuamente recursivas,

$$
\begin{cases} f_1 = \phi_1\ (f_1, \ldots, f_n) \\ \vdots \\ f_n = \phi_n\ (f_1, \ldots, f_n) \end{cases}
$$

desde que todas as funções f_i partilhem o mesmo functor de base. Por exemplo, para $n = 3$, obtém-se:

$$\begin{cases} f \cdot in = i \cdot \mathbf{F}\ \langle f, \langle g, h\rangle\rangle \\ g \cdot in = j \cdot \mathbf{F}\ \langle f, \langle g, h\rangle\rangle \\ h \cdot in = k \cdot \mathbf{F}\ \langle f, \langle g, h\rangle\rangle \end{cases} \equiv \langle f, \langle g, h\rangle\rangle = (\!|\, \langle i, \langle j, k\rangle\rangle \,|\!)$$

Em 4.2, estudar-se-á um exemplo prático de recursividade mútua. Mas antes, vamos introduzir a lei de "banana-split", um corolário da lei de recursividade mútua.

4.1 "BANANA-SPLIT"

Seja $f = (\!|\, i \,|\!)$ e $g = (\!|\, j \,|\!)$. Então,

$$\begin{aligned} & f = (\!|\, i \,|\!) \\ \equiv & \quad \{ \text{ universal-cata } \} \\ & f \cdot in = i \cdot \mathbf{F}\ f \\ \equiv & \quad \{ \text{ cancelamento-} \times \} \\ & f \cdot in = i \cdot \mathbf{F}\ (\pi_1 \cdot \langle f, g\rangle) \\ \equiv & \quad \{ \text{ F é um functor } \} \\ & f \cdot in = i \cdot \mathbf{F}\ \pi_1 \cdot F\langle f, g\rangle \end{aligned}$$

Similarmente,

$$\begin{aligned} & g = (\!|\, j \,|\!) \\ \equiv & \quad \{ \text{ tal como em cima para } g = (\!|\, j \,|\!) \} \\ & g \cdot in = j \cdot \mathbf{F}\ \pi_2 \cdot F\langle f, g\rangle \end{aligned}$$

Por fim,

$$\begin{cases} f \cdot in = i \cdot \mathbf{F}\ \pi_1 \cdot \mathbf{F}\ \langle f, g \rangle \\ g \cdot in = j \cdot \mathbf{F}\ \pi_2 \cdot \mathbf{F}\ \langle f, g \rangle \end{cases} \equiv \langle f, g \rangle = (\!|\ \langle h, k \rangle\ |\!)$$

$$\equiv \quad \{ \text{ lei de recursividade mútua } \}$$

$$\langle f, g \rangle = (\!|\ \langle i \cdot \mathbf{F}\ \pi_1, j \cdot \mathbf{F}\ \pi_2 \rangle\ |\!)$$

$$\equiv \quad \{\ f = (\!|\, i \,|\!) \text{ e } g = (\!|\, j \,|\!)\ \}$$

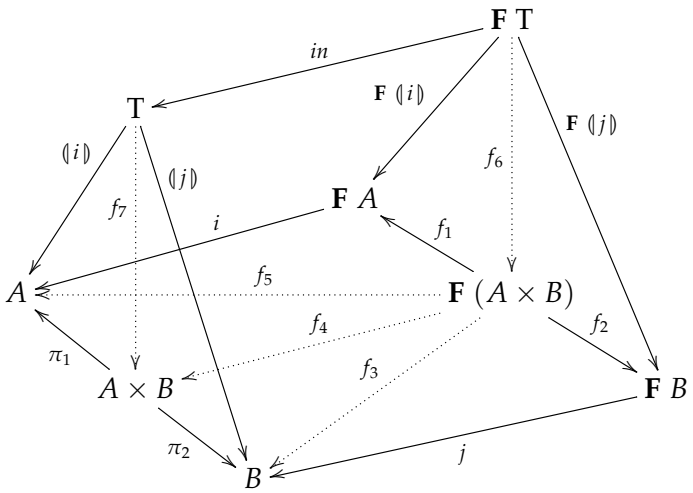
$$\langle (\!|\, i \,|\!), (\!|\, j \,|\!) \rangle = (\!|\ \langle i \cdot \mathbf{F}\ \pi_1, j \cdot \mathbf{F}\ \pi_2 \rangle\ |\!)$$

de onde surge, através da propriedade de *absorção-*$\times$, a lei de "banana-split":

$$\langle (\!|\, i \,|\!), (\!|\, j \,|\!) \rangle = (\!|\ (i \times j) \cdot \langle \mathbf{F}\ \pi_1, \mathbf{F}\ \pi_2 \rangle\ |\!) \tag{62}$$

A lei de "banana-split" é muito útil na programação funcional pois permite combinar dois "ciclos", $(\!|\, i \,|\!)$ e $(\!|\, j \,|\!)$, num único "ciclo" $(\!|\ \langle i \cdot \mathbf{F}\ \pi_1, j \cdot \mathbf{F}\ \pi_2 \rangle\ |\!)$.

É representada pelo seguinte diagrama:



Para completar o diagrama, basta definir f_i, $i \in \{1,2,3,4,5,6,7\}$ do seguinte modo:

$$
\begin{aligned}
f_1 &= \mathbf{F}\ \pi_1 \\
f_2 &= \mathbf{F}\ \pi_2 \\
f_3 &= j \cdot \mathbf{F}\ \pi_2 \\
f_4 &= \langle i \cdot \mathbf{F}\ \pi_1, j \cdot \mathbf{F}\ \pi_2 \rangle \\
f_5 &= i \cdot \mathbf{F}\ \pi_1 \\
f_6 &= \mathbf{F}\ \langle (\!|\, i \,|\!), (\!|\, j \,|\!) \rangle \\
f_7 &= \langle (\!|\, i \,|\!), (\!|\, j \,|\!) \rangle
\end{aligned}
$$

4.2 SEQUÊNCIA DE FIBONACCI

O poder da recursividade mútua pode ser visto em vários exemplos. Um deles é no cálculo dos números de Fibonacci.

Em matemática, os números de Fibonacci, geralmente denotados por F_n, formam uma sequência, chamada *Sequência de Fibonacci*, onde cada número é obtido a partir da soma dos dois anteriores, isto é,

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Esta sequência encontra-se muitas vezes representada na natureza, através da espiral obtida pela junção dos quadrados cujos comprimentos dos lados correspondem aos valores da sequência.

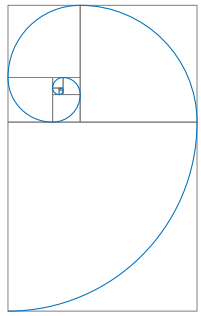


Figura 1: Espiral resultante da sequência de Fibonacci.

A sequência F_n é então definida do seguinte modo:

$$
\begin{aligned}
F_0 &= 1 \\
F_1 &= 1 \\
F_{n+1} &= F_n + F_{n-1},\ n > 1
\end{aligned}
$$

Para programar esta sequência, podemos usar a fórmula dada em cima. No entanto, ela é muito ineficiente! Existe uma degradação exponencial no seu tempo de execução. Num computador normal, demoraria várias horas para calcular, por exemplo, o quinquagésimo número da sequência!

Graças à recursividade mútua, é possível reduzir esta duração para apenas alguna segundos! Assim sendo, vamos definir a função *fibonacci* a partir da lei de recursividade mútua!

A versão original pode ser reescrita em HASKELL da seguinte forma:

```
fib 0 = 1
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

Vamos agora recorrer à lei de recursividade mútua para obter uma solução muito mais eficiente.

Uma vez que estamos perante o functor dos naturais,

$$
\begin{aligned}
&in = [\underline{0}, succ]\\
&out\ 0 = i_1\ ()\\
&out\ (n+1) = i_2\ n\\
&\mathbf{F}\ X = 1 + X\\
&\mathbf{F}\ f = id + f
\end{aligned}
$$

Assim, com base na nota, vamos proceder ao cálculo da função *fibonacci*.

Parte-se de 61.

$$
\begin{cases} f \cdot in = h \cdot \mathbf{F}\ \langle f, g\rangle \\ g \cdot in = k \cdot \mathbf{F}\ \langle f, g\rangle \end{cases} \quad \equiv \langle f, g\rangle = (\!|\ \langle h, k\rangle\ |\!)
$$

$\equiv$ { definição de in; functor dos naturais }

$$
\begin{cases} f \cdot [\underline{0}, succ] = h \cdot (id + \langle f, g\rangle) \\ g \cdot [\underline{0}, succ] = k \cdot (id + \langle f, g\rangle) \end{cases} \quad \equiv \langle f, g\rangle = (\!|\ \langle h, k\rangle\ |\!)
$$

$\equiv$ { fusão-+; reflexão-+; eq-+; igualdade extensional;

def-comp; def-split; def-$\times$ }

$$
\begin{cases} \begin{cases} f\ 0 = a \\ f\ (n+1) = h_2\ (f\ n, g\ n) \end{cases} \\ \begin{cases} g\ 0 = b \\ g\ (n+1) = k_2\ (f\ n, g\ n) \end{cases} \end{cases} \quad \equiv \langle f, g\rangle = (\!|\ \langle [\underline{a}, h_2], [\underline{b}, k_2]\rangle\ |\!) \tag{63}
$$

Vamos fazer uma mudança de variável no programa original:

```
fib (n+2) = f (n+1)
f (n+1) = f n + fib n
fib (n+1) = f n
```

A partir disto, podemos reescrever o programa da seguinte forma:

```
f 0 = 1
f (n+1) = f n + fib n
fib 0 = 1
fib (n+1) = f n
```

e a partir daqui e de 63 obtemos:

$$
\begin{aligned}
&h_2 = add \\
&k_2 = \pi_1 \\
&a = b = 1 \\
&g \text{ renomeada para } fib
\end{aligned}
$$

$$
\begin{aligned}
\langle f, fib\rangle &= (\!|\, \langle [\underline{1}, add\,], [\underline{1}, \pi_1]\rangle \,|\!) \\
fibonacci &= \pi_2 \cdot \langle f, fib\rangle
\end{aligned}
$$

Adicionalmente, podemos definir a função *fibonacci* numa linguagem imperativa, por exemplo C, à custa da *lei da troca* e a definição do ciclo *for* como vemos em seguida.

$$
\begin{aligned}
\langle f, fib\rangle &= (\!|\, \langle [\underline{1}, add\,], [\underline{1}, \pi_1]\rangle \,|\!) \\
\langle f, fib\rangle &= \text{ for } \langle add, \pi_1\rangle\ (1,1) \\
fibonacci &= \pi_2 \cdot (\text{for } \langle add, \pi_1\rangle\ (1,1))
\end{aligned}
$$

De acordo com as seguintes sugestões:

- O corpo do ciclo deve ter tantos argumentos quanto o número de funções mutuamente recursivas;
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente;[1]
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável n;

1 Podem obviamente usar-se outros símbolos, mas numa primeira leitura é mais intuitivo usarem-se tais nomes.

- Em *init* coleccionam-se os resultados dos casos de base das funções, pela mesma ordem.

obtemos de imediato a função *fibonacci*, na linguagem imperativa C:

```
int fibonacci(int n){
    int x = 1, y = 1, i;
    for (i = 1; i <= n; i++){
        int a = x;
        x = x + y;
        y = a;
    }
    return y;
}
```

5

ANAMORFISMOS

Um *anamorfismo* pode ser definido, de uma forma informal, como o "contrário" de um catamorfismo. Voltemos ao nosso reduzido conhecimento da antiga língua grega.

Anamorfismo deriva de *ana* "para cima" + *morphe* "forma". Novamente, a etimologia da palavra permite-nos deduzir o comportamento de um *anamorfismo* – uma função que "constrói" (ou "levanta") uma determinada estrutura de dados.

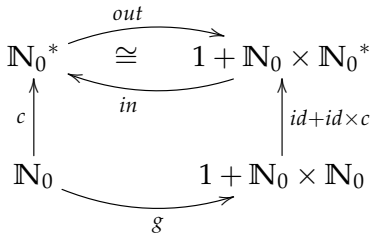
Tal como para os catamorfismos, vamos recorrer a um exemplo.

5.1 ANAMORFISMO SOBRE LISTAS

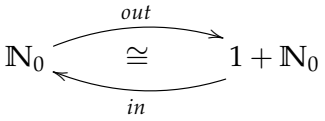
Como exemplo, vamos considerar a função c que, dado um $n \in \mathbb{N}$, constrói a lista $[1..n]$ invertida (ex.: $c\ 4 = [4, 3, 2, 1]$). Pode ser escrita em HASKELL da seguinte forma:

```
c 0 = []
c n = n : c (n-1)
```

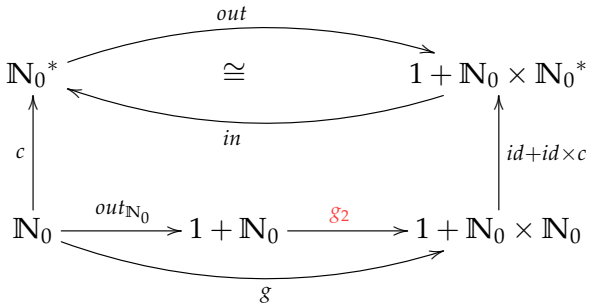
Vamos começar por analisar o seguinte diagrama e tentar definir a função g:



Relembremos o isomorfismo de construção de naturais:



A função *out* é muito útil para a definição de g, pelo que a vamos inserir no diagrama com o nome $out_{\mathbb{N}_0}$ para não confundir com a função *out* para listas já presente no diagrama.

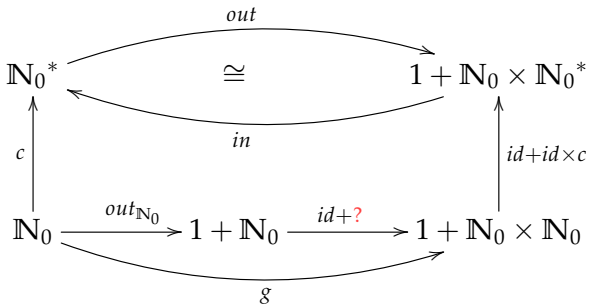


Assim, podemos já definir $g = g_2 \cdot out_{\mathbb{N}_0}$, faltando apenas definir g_2 que será, evidentemente, uma soma de funções.

Sabendo que um natural ou é 0 ou é o sucessor de um número natural, vamos abordar ambos os casos separados.

Se a função g receber um 0, a função $out_{\mathbb{N}_0}$ injeta $()$ à esquerda. Reparemos que, no canto inferior direito, o tipo da esquerda é 1. Ou seja, existe o único habitante desse tipo, $()$, é preservado pela

identidade na segunda seta vertical, e o *in* das listas produz uma lista vazia – que é o esperado para $g\ 0$. Então,



Por fim, ? refere-se, naturalmente, ao segundo e último caso – quando g recebe o **sucessor** de um número natural.

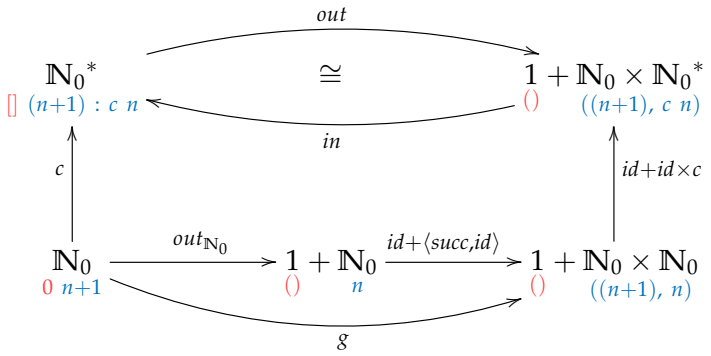
Quando g recebe um valor $n+1$, a função $out_{\mathbb{N}_0}$ injeta o valor n à direita.[1] O objetivo é aplicar recursivamente a função c ao valor n, e, no final, construir a lista $((n+1)\ :\ c\ n)$. Ora, tendo o valor n, basta apenas formar o par $(n+1, n)$, visto que na segunda seta vertical, $n+1$ é preservado pela identidade e é aplicada a função c ao n, formando o par $((n+1), c\ n)$, o qual é passado como argumento à função *in* das listas, construindo a lista $((n+1)\ :\ c\ n)$, como pretendido!

Concluímos que $g = (id + \langle succ, id\rangle) \cdot out_{\mathbb{N}_0}$ e diremos que g é o *gene* do anamorfismo c, usando a seguinte notação:

$$c = [\![g]\!] = [\![(id + \langle succ, id\rangle) \cdot out_{\mathbb{N}_0}]\!]$$

1 Não injeta $n+1$! Por exemplo, caso se execute $g\ 23$, então $out_{\mathbb{N}_0}$ injeta 22 à direita.

É possível ver, no diagrama seguinte, uma pequena representação do que acontece em cada caso. A vermelho está o caso $g\ 0$ e a azul $g\ (n+1)$.



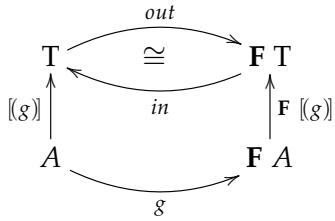
$$
\begin{aligned}
& c = in_L \cdot (id + id \times c) \cdot (id + \langle succ, id \rangle) \cdot out_{\mathbb{N}_0} \\
\equiv & \quad \{ \text{ def-}in_L \text{; isomorfismo } in \text{ e } out \ \} \\
& c \cdot in_{\mathbb{N}_0} = [nil, cons] \cdot (id + id \times c) \cdot (id + \langle succ, id \rangle) \\
\equiv & \quad \{ \text{ def-}in_{\mathbb{N}_0} \text{; absorção-+ 2x } \} \\
& c \cdot [\underline{0}, succ] = [nil, cons \cdot (id \times c) \cdot \langle succ, id \rangle] \\
\equiv & \quad \{ \text{ fusão-+; eq-+; absorção-}\times \text{; natural-}id \ \} \\
& \begin{cases} c \cdot \underline{0} = nil \\ c \cdot succ = cons \cdot \langle succ, c \rangle \end{cases} \\
\equiv & \quad \{ \text{ igualdade extensional; def-comp; def-}split \text{; def-}succ \ \} \\
& \begin{cases} c\ 0 = [\,] \\ c\ (n+1) = (n+1)\ :\ c\ n \end{cases}
\end{aligned}
$$

Tal como fizemos para o combinador catamorfismo, obtivemos exatamente a mesma definição da função c apresentada no início da secção.

Novamente, reconhecemos o functor das listas presente na parte recursiva do diagrama. Assim, podemos também generalizar o diagrama de um *anamorfismo* para um qualquer tipo indutivo T.

5.2 GENERALIZAÇÃO

Diagrama geral de um anamorfismo $[\![(g)]\!]$ para um tipo indutivo T:



5.3 PROPRIEDADES

- Universal-ana:

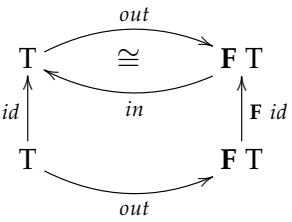
$$k = [\![(g)]\!] \equiv out \cdot k = (\mathbf{F}\ k) \cdot g \tag{64}$$

- Cancelamento-ana:

$$out \cdot [\![(g)]\!] = \mathbf{F}\ [\![(g)]\!] \cdot g \tag{65}$$

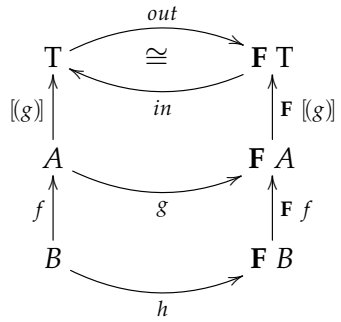
- Reflexão-ana:

$$[\![(out)]\!] = id_{\mathrm{T}} \tag{66}$$



- Fusão-ana:

$$[\![(g)]\!] \cdot f = [\![(h)]\!] \Leftarrow g \cdot f = \mathbf{F}\ f \cdot h \tag{67}$$



- Absorção-ana:

$$\mathbf{T}\ f \cdot [\![(g)]\!] = [\![(\mathbf{B}(f, id) \cdot g)]\!] \tag{68}$$

6

HILOMORFISMOS

Uma vez estudado o conceito de *catamorfismo* e *anamorfismo*, podemos compor estes dois combinadores, criando uma função que inicialmente "constrói" uma estrutura (*anamorfismo*) e depois "consome" a estrutura criada (*catamorfismo*).

Vamos usar como exemplo a função *fact* que calcula o fatorial de um $n \in \mathbb{N}_0$.

Esta função não pode ser expressa à custa de um catamorfismo porque a estrutura dos naturais não é suficiente para produzir o fatorial de um natural. Também não pode ser expressa à custa de um anamorfismo pois um natural não é uma estrutura de dados.

Relembremos que $n! = n \times (n-1) \times (n-2) \times \ldots \times 1$ e que $0! = 1$. Portanto podemos definir a função *fact* da seguinte forma:

$$\begin{aligned} &\mathit{fact}\ 0 = 1 \\ &\mathit{fact}\ (n+1) = (n+1) \times \mathit{fact}\ n \end{aligned}$$

Então, é necessário criar uma estrutura que contenha todos os valores a multiplicar, e depois consumir a estrutura, multiplicando todos os valores nela presentes.

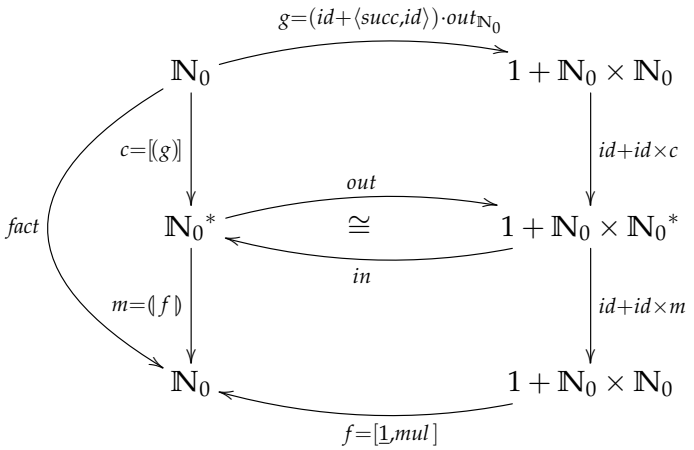
Vamos para isso recorrer ao *anamorfismo c* estudado em 5.1, que dado um $n \in \mathbb{N}_0$ calcula a lista $[1..n]$ invertida, e ao *catamorfismo*

m, que dada uma lista de naturais, retorna a multiplicação de todos os seus elementos.

Assim, a função *fact* será definida do seguinte modo:

$$fact = m \cdot c$$

e pode ser facilmente interpretada a partir do seguinte diagrama:



A notação a ser usada é a seguinte:

$$fact = [\![f, g]\!] = (\!| f |\!) \cdot (\!|g|\!) = m \cdot c$$

Vamos agora proceder ao cálculo analítico da função *fact* com base no diagrama apresentado.

$$
\begin{aligned}
& fact = f \cdot (id + id \times m) \cdot (id + id \times c) \cdot g \\
\equiv & \quad \{ \text{ def-}f\text{; def-}g\text{; functor-}+\text{; functor-}\times \ \} \\
& fact = [\underline{1}, mul\,] \cdot (id + id \times (m \cdot c)) \cdot (id + \langle succ, id \rangle) \cdot out_{\mathbb{N}_0} \\
\equiv & \quad \{ \text{ isomorfismo } in \text{ e } out\text{; functor-}+ \ \} \\
& fact \cdot in_{\mathbb{N}_0} = [\underline{1}, mul\,] \cdot (id + id \times (m \cdot c) \cdot \langle succ, id \rangle) \\
\equiv & \quad \{ \text{ def-}in_{\mathbb{N}_0}\text{; absorção-}\times\text{; natural-}id \text{ 2x } \} \\
& fact \cdot [\underline{0}, succ\,] = [\underline{1}, mul\,] \cdot (id + \langle succ, (m \cdot c) \rangle) \\
\equiv & \quad \{ \text{ fusão-}+\text{; absorção-}+\text{; natural-}id \ \} \\
& [fact \cdot \underline{0}, fact \cdot succ\,] = [\underline{1}, mul \cdot \langle succ, (m \cdot c) \rangle] \\
\equiv & \quad \{ \text{ eq-}+\text{; } m = fact \ \} \\
& \begin{cases} fact \cdot \underline{0} = \underline{1} \\ fact \cdot succ = mul \cdot \langle\, succ, fact\,\rangle \end{cases} \\
\equiv & \quad \{ \text{ igualdade extensional; def-comp; } \} \\
& \begin{cases} fact\ (\underline{0}\ n) = \underline{1}\ n \\ fact\ (succ\ n) = mul\ (\langle\, succ, fact\,\rangle\ n) \end{cases} \\
\equiv & \quad \{ \text{ def-const; def-}split\text{; def-}succ\text{; def-}mul \ \} \\
& \begin{cases} fact\ 0 = 1 \\ fact\ (n+1) = (n+1) \times fact\ n \end{cases}
\end{aligned}
$$

Como esperado, obtivemos exatamente a mesma definição da função *fact* escrita no início do capítulo.

Por fim, fazendo uma última referência à abrangente e completa clássica língua grega, *hilomorfismo* deriva de: *hylo* "matéria, coisa,

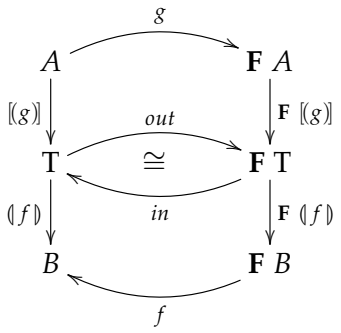
tudo" + *morphe* "forma". Isto sugere que o *hilomorfismo* é um esquema geral, ou seja, qualquer programa que possamos escrever é uma composição de um *catamorfismo* com um *anamorfismo*.

Os próprios catamorfismos e anamorfismos são casos particulares de um hilomorfismo.

$$(\!|f|\!) = [\![f, out]\!] = (\!|f|\!) \cdot [\!(out)\!] = (\!|f|\!) \cdot id = (\!|f|\!)$$
$$[\!(g)\!] = [\![in, g]\!] = (\!|in|\!) \cdot [\!(g)\!] = id \cdot [\!(g)\!] = [\!(g)\!]$$

6.1 GENERALIZAÇÃO

$$[\![f, g]\!] = (\!|f|\!) \cdot [\!(g)\!] \tag{69}$$



6.2 DIVIDE AND CONQUER

Divide & Conquer é a estratégia mãe da programação. Um algoritmo para um determinado problema não tem outra solução senão dividir o problema em partes, ou subproblemas, resolver os subproblemas e juntar as soluções, obtendo uma solução para o problema inicial.

Assim, *hilomorfismo = divide & conquer*.

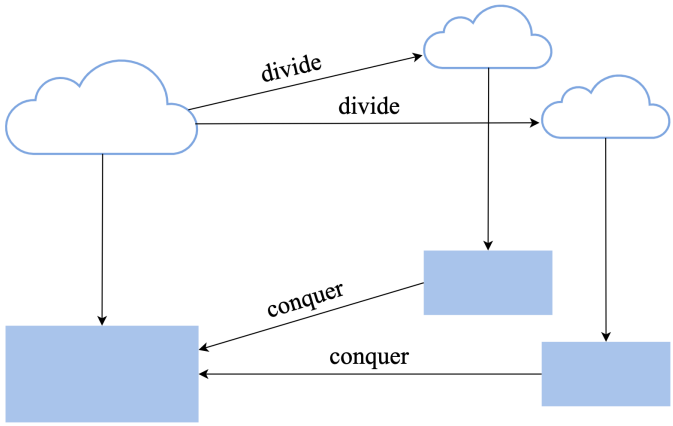


Figura 2: Ilustração da estratégia *Divide & Conquer*

Naturalmente, existem hilomorfismos "muito bons" a dividir, e por isso "poupam" na parte da conquista[1], e existem hilomorfismos que "pouco fazem" na divisão, e por isso têm de compensar na conquista.

É muito intuitivo ver que a parte da divisão do problema (*divide*) corresponderá ao *anamorfismo* e a parte da conquista das várias soluções (*conquer*) corresponderá ao *catamorfismo*.

$$algorithm = (\!|\, conquer \,|\!) \cdot [\!(divide)\!]$$

Vamos ver um exemplo.

6.2.1 *Algoritmo de Ordenação QuickSort*

O algoritmo de ordenação *QuickSort* é definido em HASKELL do seguinte modo:

```
qSort [] = []
qSort (h:t) = qSort x1 ++ [h] ++ qSort x2
```

1 Em 6.3, vamos ver um hilomorfismo particular, cujo anamorfismo praticamente produz a solução pretendida.

```
    where
        x1 = [a | a <- t, a < h]
        x2 = [a | a <- t, a > h]
```

O seguinte esquema dá-nos uma ideia de como funciona o algoritmo:

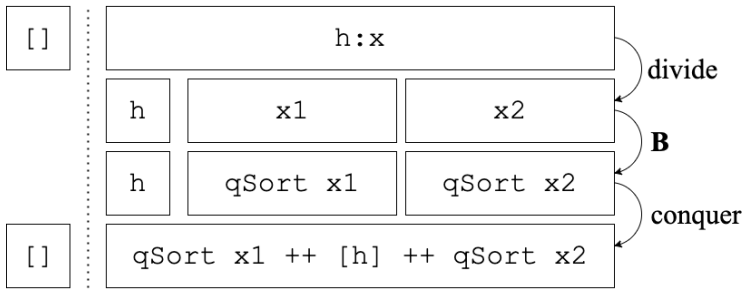


Figura 3: *Estrutura do QuickSort*

A partir do esquema, podemos desde já inferir o tipo do *divide*:

$$A^* \xrightarrow{divide} 1 + A \times (A^*)^2$$

e o bi-functor de base **B** do algoritmo:

$$\mathbf{B}(X, Y) = 1 + X \times Y^2$$

Vamos agora definir o algoritmo ao nível *pointfree*.

A partir da definição dada em HASKELL, podemos reescrever o algoritmo da seguinte forma:

$$\begin{cases} qSort \cdot nil = nil \\ qSort \cdot cons = f \cdot (id \times qSort^2) \cdot g \end{cases}$$

onde

$$
\begin{aligned}
g\ (h, x) &= (h, (x_1, x_2)) \\
&\quad \textbf{where} \\
&\quad\ x_1 = [\, a \mid a \leftarrow x,\ a < h \,] \\
&\quad\ x_2 = [\, a \mid a \leftarrow x,\ a \geq h \,]
\end{aligned}
$$

$$f\ (h, (y_1, y_2)) = y_1 +\!\!+ [h] +\!\!+ y_2$$

Para evidenciar a arquitetura *divide & conquer* do algoritmo, é necessário desdobrar a definição do seguinte modo:

$$
\begin{cases}
qSort \cdot nil = nil \\
qSort \cdot cons = f \cdot (id \times qSort^2) \cdot g
\end{cases}
$$

$$\equiv \qquad \{ \text{ fusão-+; absorção-+; eq-+; etc. } \}$$

$$qSort \cdot in = [nil, f] \cdot (id + id \times qSort^2) \cdot (id + g)$$

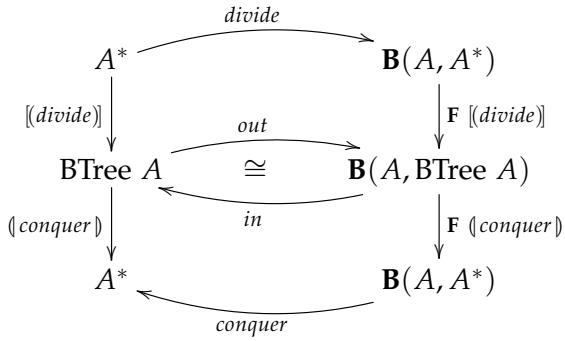
$$\equiv \qquad \{ \text{ isomorfismo } in \text{ e } out \ \}$$

$$qSort = \underbrace{[nil, f]}_{conquer} \cdot \underbrace{(id + id \times qSort^2)}_{\mathbf{B}(id, qSort)} \cdot \underbrace{(id + g) \cdot out}_{divide}$$

Como já é do nosso conhecimento, um hilomorfismo usa uma estrutura de dados intermédia. Qual é essa estrutura neste caso em particular? Para responder a esta pergunta, temos observar o bi-functor que inferimos inicialmente.

$$
\begin{aligned}
&\mathbf{B}(X, Y) = 1 + X \times Y^2 \\
&\mathbf{B}(f, g) = id + f \times g^2 \\
&\mathbf{F}\ f = \mathbf{B}(id, f)
\end{aligned}
$$

Ora, o bi-functor em causa é o bi-functor do tipo indutivo *BTree*. Isto diz-nos que o algoritmo de ordenação *QuickSort* usa uma árvore binária como estrutura intermédia.[2]

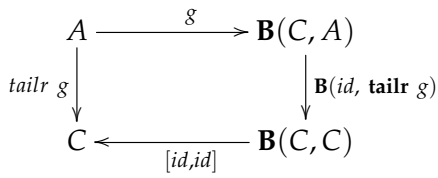


Por fim,

$$\begin{aligned} quickSort &= [\![\, conquer, divide \,]\!] \\ &= [\![\, [nil, f], (id + g) \cdot out \,]\!] \\ &= (\!|\, [nil, f] \,|\!) \cdot [\!(\, (id + g) \cdot out \,)\!] \end{aligned}$$

6.3 TAIL RECURSION

Como já referido, este hilomorfismo é um caso particular cujo anamorfismo (*divide*) praticamente produz a solução. O catamorfismo (*conquer*) apenas "retira" a solução, já produzida, da estrutura intermédia.

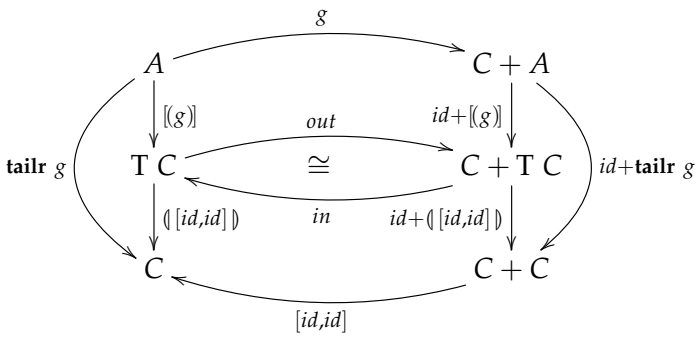


$$\mathbf{B}(X, Y) = X + Y$$

2 Na verdade, podemos ir mais além, o algoritmo de ordenação *QuickSort* usa uma árvore binária **de procura** (uma árvore bi-ordenada) como estrutura intermédia.

Qual é o tipo do hilomorfismo? Ora, o tipo do hilomorfismo terá de ter a seguinte estrutura:

$$\mathrm{T}\ X \cong \mathbf{B}(X, \mathrm{T}\ X) = X + \mathrm{T}\ X$$



Se T C deixar de ser paramétrico, o tipo do hilomorfismo passa a ser $\mathrm{T} \cong 1 + \mathrm{T}$. Este tipo de dados corresponde ao tipo dos números naturais.

Podemos então olhar para T C como um tipo semelhante ao tipo dos números naturais, com a diferença de que guarda informação no 0. Assim, T C, além de conter a solução C, contém também o número de iterações feitas.[3]

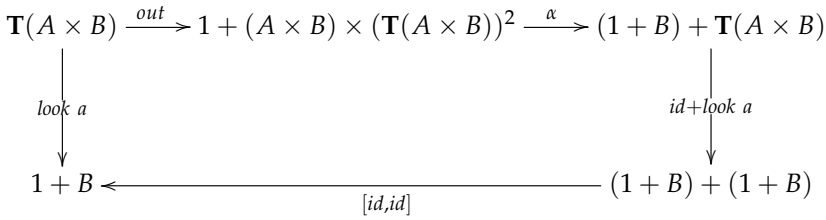
Por fim, apresenta-se em seguida um exemplo de algoritmos *Tail Recursion*.

6.3.1 *Algoritmo de Pesquisa Binária*

```
lookBTree :: ord a => a -> BTree(a, b) -> Maybe b
lookBTree a Empty = Nothing
lookBTree a (Node((a', b'), (l, r)))
        | a == a' = Just b'
```

3 É possível ver esta implementação, em HASKELL, no instante t=10:43 da aula T10b [1].

```
  | a < a' = lookBTree a l
  | a > a' = lookBTree a r
```



$$
\begin{aligned}
&\alpha = [i_1 \cdot i_1, \mathit{decide}\ a] \\
&\mathit{decide}\ a\ ((a', b'), (l, r)) \\
&\qquad \mid a == a' = i_1\ (i_2\ b') \\
&\qquad \mid a < a' = i_2\ l \\
&\qquad \mid a > a' = i_2\ r
\end{aligned}
$$

7

MÓNADES

Mónades são functores de corrida

J.N. Oliveira

De forma informal, podemos definir um mónade como um functor com propriedades especiais. Para entender a importância destes "functores de corrida", devemos começar por estudar o que são funções parciais. Este será o nosso ponto de partida para entender o que é um mónade.

7.1 FUNÇÕES PARCIAIS

Uma função parcial é uma função que não está definida para todos os elementos do seu domínio, e, portanto, nem sempre consegue produzir um *output*.

Por exemplo, sabendo que a função *head* retorna o primeiro elemento de uma lista, o que é que acontece quando "interrogamos" o GHCi com

```
> head []
```

?

Ocorre um erro, porque a lista vazia não tem nenhum elemento, por isso não é possível retornar o primeiro elemento, pois ele não existe. Contudo, é possível reescrever a função *head* de forma a cobrir este caso. Uma maneira de o fazer seria por tirar partido do tipo *Maybe*.

```
head [] = Nothing
head (h:t) = Just h
```

Uma vez que $Maybe\ A \cong 1 + A$, a nova função *head* tem o tipo $1 + A \xleftarrow{head} A^*$.

Desta forma, conseguimos cobrir todos os casos de uma lista, fechando a porta ao erro de há pouco. Este exemplo é simples, mas com ele conseguimos ver o quão importante são as funções parciais, para impedir que erros aconteçam.

Generalizando, obtemos, para qualquer função $B \xleftarrow{f} A$, a seguinte função g:

$$1 + B \xleftarrow{g=i_2 \cdot f} A$$

Ora, para compor uma função, digamos h, com g, é necessário que h tenha como tipo de entrada $1 + B$.

$$C \xleftarrow{h} 1 + B \xleftarrow{g=i_2 \cdot f} A$$

Questão: e se a função h não tem como tipo de entrada $1 + B$, mas apenas B?

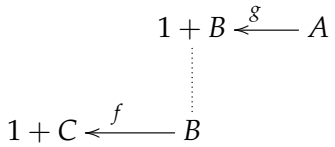
Questão: é possível compor funções parciais?

7.2 COMPOSIÇÃO DE FUNÇÕES PARCIAIS

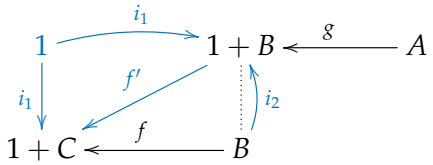
Sejam f e g as seguintes funções:

$$1 + C \xleftarrow{f} B$$
$$1 + B \xleftarrow{g} A$$

Como é que vamos compor as duas funções?



Não podemos simplesmente fazer $f \cdot g$, visto que f requer um valor do tipo B e a função g retorna um valor do tipo $1 + B$. Então, para compor funções parciais, vamos ter de utilizar uma função que está "escondida" no diagrama.



(70)

f' é a nossa função escondida que irá possibilitar a composição de funções parciais. Uma vez que $f' = [i_1, f]$, a definição da composição de funções parciais é dada do seguinte modo:

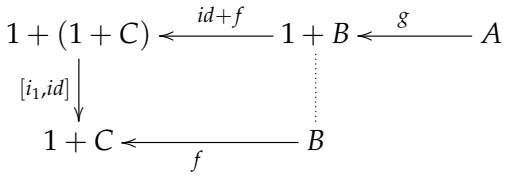
$$f \bullet g \stackrel{\text{def}}{=} [i_1, f] \cdot g \tag{71}$$

7.3 COMPOSIÇÃO DE KLEISLI

A partir do diagrama 70, podemos inferir a definição de f' como sendo:

$$\begin{aligned} & f' = [i_1, f] \\ \equiv & \{\text{absorção-+}\} \\ & f' = [i_1, id] \cdot (id + f) \end{aligned}$$

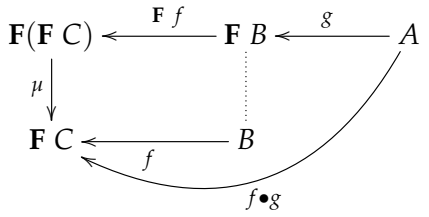
e definir o seguinte diagrama:



Reparemos que, definindo o seguinte functor:

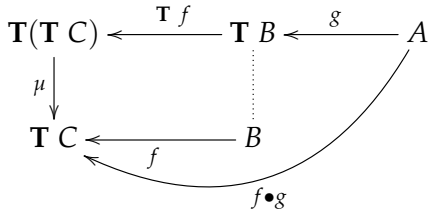
$$\begin{aligned} \mathbf{F}\ f &= id + f \\ \mathbf{F}\ X &= 1 + X \end{aligned}$$

e definido a função $\mu = [i_1, id]$, obtemos o seguinte diagrama:



Conforme vimos em 1.11, um functor **F** transforma um tipo de dados X numa estrutura mais elaborada que contém X's. Além disso, estudamos também tipos indutivos paramétricos, $\mathbf{T}\ X$, cuja estrutura mais elaborada era definida à custa do bi-functor de base $\mathbf{B}(X, \mathrm{T}X)$.

Assim, podemos generalizar o diagrama da *composição de Kleisli* para um qualquer tipo indutivo paramétrico $\mathbf{T}\ X$ e uma função $\mu : \mathbf{T}\ C \leftarrow \mathbf{T}(\mathbf{T}\ C)$.



Estamos agora prontos para entender o que é um mónade.

7.4 O QUE É UM MÓNADE?

Um mónade é um functor que dispõe de duas funções, u (unidade) e μ (multiplicação), cujos tipos são:

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}(\mathbf{T}\ X)$$

e obedece às duas seguintes propriedades:

$$\mu \cdot u = \mu \cdot \mathbf{T}\ u = id \tag{72}$$

$$\mu \cdot \mu = \mu \cdot \mathbf{T}\ \mu \tag{73}$$

7.4.1 *Exemplo: Mónade LTree*

Uma *Leaf Tree* pode ser definida em HASKELL do seguinte modo:

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

O *isomorfismo* de construção de *Leaf Trees* é o seguinte:

$$\text{LTree } A \underset{in=[Leaf,Fork]}{\overset{out}{\cong}} A + (\text{LTree } A)^2$$

Como estudamos, um mónade dispõe de duas funções, u e μ.

$$A \xrightarrow{u} \text{LTree } A \xleftarrow{\mu} \text{LTree } (\text{LTree } A)$$

A função u é explicita na definição do tipo de dados *LTree*.

$$u = Leaf$$

Falta apenas definir μ. Para isso, vamos recorrer ao combinador *catamorfismo*.

$$\begin{array}{ccc} \text{LTree } (\text{LTree } A) & \overset{out}{\underset{in=[Leaf,Fork]}{\cong}} & \text{LTree } A + (\text{LTree } (\text{LTree } A))^2 \\ \downarrow{\scriptstyle \mu} & & \downarrow{\scriptstyle id+\mu^2} \\ \text{LTree } A & \xleftarrow[[id,Fork]]{} & \text{LTree } A + (\text{LTree } A)^2 \end{array}$$

Concluímos que $\mu = (\!|\, [id, Fork] \,|\!)$. Por fim, é necessário verificar as leis 72 (unidade) e 73 (multiplicação).

- *unidade*: $\mu \cdot u = \mu \cdot \mathbf{T}\, u = id$

Comecemos por mostrar que $\mu \cdot \mathbf{T}\ u = id$.

$$
\begin{array}{ll}
& \mu \cdot \mathbf{T}\ u \\
= & \quad \{\ \mu = (\!|\, [id, Fork]\, |\!);\ u = Leaf\ \} \\
& (\!|\, [id, Fork]\, |\!) \cdot \text{LTree}\ Leaf \\
= & \quad \{\ \text{absorção-cata}\ \} \\
& (\!|\, [id, Fork]\, \cdot (Leaf + id^2)\, |\!) \\
= & \quad \{\ \text{absorção-}+,\ \text{functor-}id\text{-}\times;\ \text{natural-}id\} \\
& (\!|\, [Leaf, Fork]\, |\!) \\
= & \quad \{\ \text{reflexão-cata}\ \} \\
& id \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \square
\end{array}
$$

Falta mostrar que $\mu \cdot u = id$. Para isso (e para a prova da lei de *multiplicação*), é importante saber que:

$$
\begin{array}{ll}
& in = [\, Leaf, Fork\,] \\
\equiv & \quad \{\ \text{universal-}+\ \} \\
& \begin{cases} Leaf = in \cdot i_1 \\ Fork = in \cdot i_2 \end{cases} \quad \underset{\equiv}{iso\ \text{in e out}} \quad \begin{cases} out \cdot Leaf = i_1 \\ out \cdot Fork = i_2 \end{cases}
\end{array}
$$

Assim,

$$
\begin{aligned}
& \mu \cdot u \\
= \quad & \{\ \mu = (\!|\, [\, id, Fork \,] \,|\!);\ u = Leaf \,\} \\
& (\!|\, [\, id, Fork \,] \,|\!) \cdot Leaf \\
= \quad & \{\ \text{universal-cata} \,\} \\
& [\, id, Fork \,] \cdot (id + \mu^2) \cdot out \cdot Leaf \\
= \quad & \{\ \text{absorção-+; natural-id} \} \\
& [\, id, Fork \cdot \mu^2 \,] \cdot out \cdot Leaf \\
= \quad & \{\ out \cdot Leaf = i_1 \,\} \\
& [\, id, Fork \cdot \mu^2 \,] \cdot i_1 \\
= \quad & \{\ \text{cancelamento-+} \,\} \\
& id \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \square
\end{aligned}
$$

- *multiplicação*: $\mu \cdot \mu = \mu \cdot \mathbf{T}\ \mu$

$$
\begin{array}{ll}
& \mu \cdot \mu = \mu \cdot \mathbf{T}\ \mu \\
\equiv & \quad \{ \text{ definição de } \mu \text{; absorção-cata } \} \\
& \mu \cdot (\!|\, [\, id, Fork \,]\, |\!) = (\!|\, [\, id, Fork \,] \cdot (\mu + \mathbf{F}\ id)\, |\!) \\
\equiv & \quad \{ \text{ definição de } \mu \text{; absorção-+; functor-id } \} \\
& \mu \cdot (\!|\, [\, id, Fork \,]\, |\!) = (\!|\, [\, \mu, Fork \,]\, |\!) \\
\Leftarrow & \quad \{ \text{ fusão-cata } \} \\
& \mu \cdot [\, id, Fork \,] = [\, \mu, Fork \,] \cdot (id + \mu^2) \\
\equiv & \quad \{ \text{ fusão-+; abosrção-+; eq-+; definição de } \mu \} \\
& \begin{cases} \mu = \mu \\ (\!|\, [\, id, Fork \,]\, |\!) \cdot Fork = Fork \cdot \mu^2 \end{cases} \\
\equiv & \quad \{ \, Fork = in \cdot i_2 \, \} \\
& (\!|\, [\, id, in \cdot i_2 \,]\, |\!) \cdot in \cdot i_2 = in \cdot i_2 \cdot \mu^2 \\
\equiv & \quad \{ \text{ cancelamento-cata } \} \\
& [\, id, in \cdot i_2 \,] \cdot (id + \mu^2) \cdot i_2 = in \cdot i_2 \cdot \mu^2 \\
\equiv & \quad \{ \text{ absorção-+ } \} \\
& [\, id, in \cdot i_2 \cdot \mu^2 \,] \cdot i_2 = in \cdot i_2 \cdot \mu^2 \\
\equiv & \quad \{ \text{ cancelamento-+ } \} \\
& true \qquad \square
\end{array}
$$

Concluímos assim que *LTree* é um mónade.

O mónade *LTree*, tal como o mónade das listas (sequências) finitas, o mónade *BTree*, o mónade *Maybe*, entre outros, são casos particulares do seguinte mónade:

$$\mathbf{T}\ X \underset{in}{\overset{out}{\cong}} \mathbf{B}(X, \mathbf{T}\ X)$$

$$\mathbf{B}(X, Y) = X + \mathbf{F}\ Y$$

$$\mathbf{B}(f, g) = f + \mathbf{F}\ g \quad (\mathbf{F} \text{ é um functor qualquer})$$

$$\mu = (\!|\, [id, in \cdot i_2] \,|\!)$$

$$u = in \cdot i_1$$

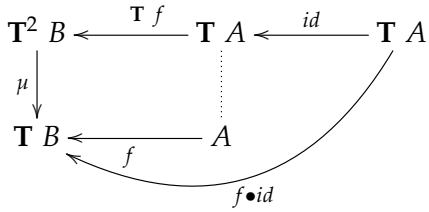
7.5 PROPRIEDADES NATURAIS

$$\begin{array}{ccccc} X & \xrightarrow{u} & \mathbf{T}\ X & \xleftarrow{\mu} & \mathbf{T}(\mathbf{T}\ X) \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathbf{T}\ f} & & \downarrow{\scriptstyle \mathbf{T}(\mathbf{T}\ f)} \\ Y & \xrightarrow[u]{} & \mathbf{T}\ Y & \xleftarrow[\mu]{} & \mathbf{T}(\mathbf{T}\ Y) \end{array}$$

$$\begin{cases} \mathbf{T}\ f \cdot u = u \cdot f \\ \mathbf{T}\ f \cdot \mu = \mu \cdot \mathbf{T}^2\ f \end{cases} \tag{74}$$

7.6 APLICAÇÃO MONÁDICA – BINDING

O que acontece quando compomos monadicamente uma função f com a identidade? Será que $f \bullet id = f$? Não! Vejamos o seguinte diagrama:

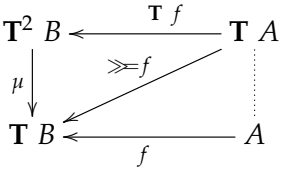


Concluímos que:

$$\begin{aligned} & f \bullet id = \mu \cdot \mathbf{T}\, f \cdot id \\ \equiv & \{ \text{ natural-}id \ \} \\ & f \bullet id = \mu \cdot \mathbf{T}\, f \end{aligned}$$

A notação a ser usada é a seguinte:

$$x \gg\!= f \stackrel{\text{def}}{=} (\mu \cdot \mathbf{T}\, f)\ x \tag{75}$$



7.7 NOTAÇÃO-DO

$$(f \bullet g)\ a = \text{do } \{\ b \leftarrow g\ a\ ;\ f\ b\ \} \tag{76}$$

$$(f \bullet id)\ x = x \gg\!= f = \text{do } \{\ b \leftarrow x\ ;\ f\ b\ \} \tag{77}$$

7.8 RECURSIVIDADE MONÁDICA

$$
\begin{aligned}
& mfor\ b\ i = (\!|\, [\underline{u}\ i, \mathbf{T}\ b] \,|\!) \\
\equiv & \\
& \begin{cases} mfor\ b\ i\ 0 = \text{return}\ i \\ mfor\ b\ i\ (n+1) = \text{do}\ \{x \leftarrow mfor\ b\ i\ n\ ;\ \text{return}(b\ x)\} \end{cases}
\end{aligned}
$$

Todos os capítulos do livro, em particular o último, ainda não estão completos.

BIBLIOGRAFIA

[1] J.N. Oliveira. Aulas teóricas de cálculo de programas. https://www4.di.uminho.pt/~jno/media/cp/.

[2] J.N. Oliveira. *Program Design by Calculation*. Universidade do Minho, Braga, 2020.