P.R. Pereira

Universidade do Minho

Cálculo de Programas

(Notas de Estudo)

Última edição: Abril, 2021

CONTEÚDO

PREÂMBULO

Este livro serve como suporte para a unidade curricular de *Cálculo de Programas*[1] dos cursos MIEI e LCC da Universidade do Minho e é baseado no livro do professor José N. Oliveira[2], regente da unidade curricular.

Universidade do Minho, Braga, Abril 2021

Paulo R. Pereira

1

FUNÇÕES E TIPOS

Vamos voltar às aulas de matemática do secundário...

1.1 O QUE É UMA FUNÇÃO?

Uma função é como uma *caixa mágica* que recebe algo e produz algo.

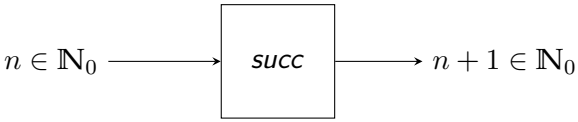
Suponhamos que temos um valor do tipo A e colocamos esse valor-A na *caixa mágica*. Alguma coisa acontece e a caixa produz algo com esse valor-A.

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 ira 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!

Assim, podemos definir a função *succ* (sucessor) como

$$succ\ n = n + 1$$

e podemos ver o seu processo no seguinte diagrama:



A notação que vamos usar para descrever funções é muito intuitiva e, neste caso, irá corresponder a:

$$\mathbb{N}_0 \xleftarrow{succ} \mathbb{N}_0$$

O objetivo, a partir de agora, é trabalhar abstratamente com funções, olhando apenas para o seu tipo (o que recebem e o que retornam). Assim, o tipo mais geral de uma função será:

$$B \xleftarrow{f} A$$

onde a função f recebe um valor-A e retorna um valor-B.

1.2 IGUALDADE FUNCIONAL

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

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

$=_B$ *denota igualdade ao nível do* B.

1 Ver A.

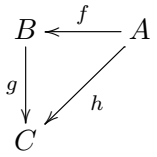
1.3 COMPOSIÇÃO FUNCIONAL

Sejam f e g as seguintes funções:

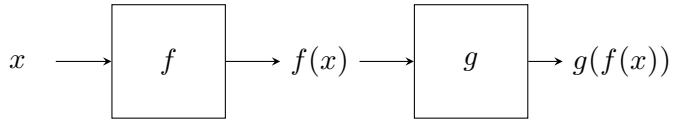
$$B \xleftarrow{\quad f \quad} A$$

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

É possível definir uma outra função, h, que recebe um valor-A e retorna um valor-C. A função h será definida como a composição de g e f como vemos em seguida.



O conceito aprendido no secundário é o mesmo.



Analiticamente, a composição é definida da seguinte forma:

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

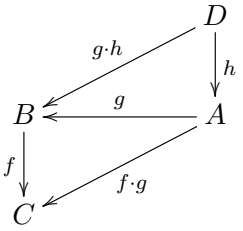
e lemos "g após f".

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

Composição de funções é o mais básico de todos os combinadores funcionais. A maioria dos programadores nem sequer pensa nela 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*:

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

A composição funcional é uma das bases desta disciplina.

1.4 FUNÇÃO IDENTIDADE

A função identidade é aquela que, no contexto de uma função, é a mais inútil. 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 e o seu tipo é expresso no seguinte diagrama:

$$A \xleftarrow{id} A$$

Assim, podemos definir a *função identidade* da seguinte forma:

$$id\ x \stackrel{\text{def}}{=} x \tag{5}$$

e a sua *propriedade natural*:

$$f \cdot id = id \cdot f = f \tag{6}$$

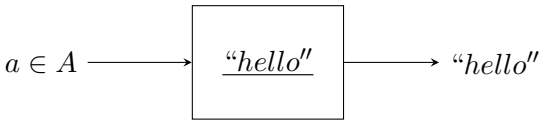
onde f é uma qualquer função.

1.5 FUNÇÕES CONSTANTES

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

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

A notação que vamos usar para descrever uma função constante é o sublinhado. No exemplo em cima, podemos definir a função como $g = const\ \text{“}hello\text{”}$, mas a notação a usar será $g = \underline{\text{“}hello\text{”}}$.



Podemos expressar o tipo da função da seguinte forma[2]:

$$S \xleftarrow{\underline{\text{“}hello\text{”}}} A$$

Assim, a definição de uma função constante é:

$$\underline{k}\ x \stackrel{\text{def}}{=} k \tag{7}$$

k é um valor arbitrário

No geral, sendo f uma qualquer função,

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

$$f \cdot \underline{k} = \underline{f\ k} \tag{9}$$

Estas 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 contradomínio.

2 **Nota**: S designa o tipo *String*. Iremos usá-lo de agora em diante.

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.

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

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 contradomínio é chamada de um *epimorfismo*. O que significa "ignorar valores" do contradomínio?

Por exemplo, a função constante $g = \underline{10}$ têm como contra-domínio 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

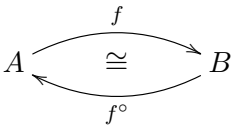
Um isomorfismo é uma função bijetiva.

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* $B \xleftarrow{f} A$ têm sempre uma função *inversa* $B \xrightarrow{f^\circ} A$.

Com a ajuda do seguinte diagrama:



podemos facilmente definir:

$$\begin{cases} f^\circ \cdot f &= id \\ f \cdot f^\circ &= id \end{cases} \tag{10}$$

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:

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

$$g \cdot f = h \equiv g = h \cdot f^\circ \tag{12}$$

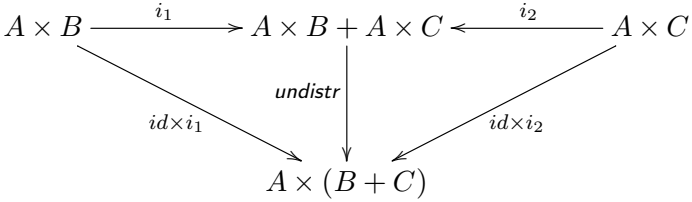
Vamos considerar um exemplo. Relembremos a *propriedade distributiva* da álgebra elementar: $x\ (y + z) = x\ y + x\ z$.

Podemos adaptar esta propriedade a tipos de dados e definir o seguinte isomorfismo:

$$A \times (B + C) \underset{\textit{undistr}}{\overset{\textit{distr}}{\cong}} A \times B + A \times C$$

onde *distr* é a função que aplica a propriedade distributiva no sentido da direita e *undistr* é a sua inversa.

Como é que a função *undistr* é definida? Bem, o seu domínio é um coproduto (uma soma), portanto a função *undistr* é definida à custa do combinador *either*.



Com a ajuda do diagrama, facilmente obtemos:

$$\textit{undistr} = [id \times i_1, id \times i_2]$$

1.8.1. Isomorfismos Relevantes

- $2 \cong \mathbb{B}$, do qual podemos inferir outro isomorfismo que será muito útil no estudo do condicional de McCarthy:

$$2 \times A \underset{\alpha}{\overset{\alpha^\circ}{\cong}} A + A \tag{13}$$

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

Nota: o tipo $\mathbb{B}$ é habitado unicamente por **2** valores: *true* e *false*.

- *Curry e Uncurry*

Seja f a seguinte função $C \xleftarrow{f} A \times B$. Podemos definir a função *curry* f, com a notação $\overline{f}$, como sendo $C^B \xleftarrow{\overline{f}} A$, onde $C^B = \{h \mid h : B \to C\}$, isto é, o conjunto das funções cujo domínio é B e o contradomínio C.

$$C^{A \times B} \underset{\text{uncurry}}{\overset{\text{curry}}{\cong}} (C^B)^A \tag{14}$$

As funções *curry* e *uncurry* estão definidas no *Standard Prelude* do HASKELL:

```
Prelude> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
Prelude> :t uncurry
uncurry :: (a -> b -> c) -> (a, b) -> c
```

Exemplo: Seja g a função que dados dois números, produz a sua soma. A versão "*uncurried*" de g, com a notação $\widehat{g}$ é:

$$\widehat{g}\ (x, y) = x + y$$

e a versão "*curried*" é:

$$\overline{g}\ x\ y = x + y$$

Por fim, podemos deduzir uma outra função, a função *ap*. Tendo um valor-A e uma função que dado um valor-A

produz um valor-B, podemos aplicá-la ao nosso valor-A e produzir um valor-B.

$$B^A \times A \xrightarrow{ap} B \tag{15}$$

E, assumindo a função $C \xrightarrow{k} B^A$, podemos definir a seguinte propriedade:

$$k = \overline{f} \equiv ap \cdot (\overline{f} \times id) = f \tag{16}$$

$$\begin{array}{ccc} B^A \times A & \xrightarrow{ap} & B \\ {\scriptstyle k\times id}\uparrow & \nearrow_{f} & \\ C \times A & & \end{array}$$

- *Álgebra de Peano*

  Qualquer elemento do conjunto $\mathbb{N}_0$ (conjunto dos números naturais) ou é 0 ou é o sucessor de um número natural.

$$\mathbb{N}_0 \underset{in}{\overset{out}{\cong}} 1 + \mathbb{N}_0 \tag{17}$$

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

1.9 PRODUTO E COPRODUTO

Em (1.3) estudamos a composição de funções. Para isso, era necessário que o domínio de uma função estivesse contido no contradomínio da outra função.

Agora, vamos estudar dois combinadores funcionais que também nos permitem "ligar" funções. Um combina funções com o mesmo domínio e o outro combina funções com o mesmo contradomínio. Vamos começar com o primeiro combinador.

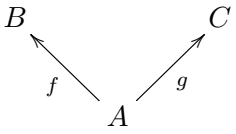
1.9.1. Combinador Split

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:

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

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

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



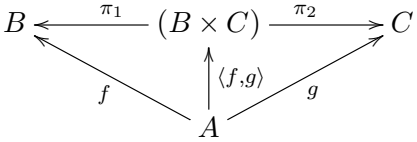
Agora, é mais intuitivo definir a função que combina f e g, produzindo um par que pertence ao Produto Cartesiano de B e C, isto é, ao seguinte conjunto:

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

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

$$\begin{array}{rcl} \langle f, g\rangle & : & A \to B \times C \\ \langle f, g\rangle\ a & \stackrel{\text{def}}{=} & (f\ a,\ g\ a) \end{array} \tag{18}$$

O seguinte diagrama expressa o seu tipo mais geral:



Nota: 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-*$\times$:

$$\pi_1 \cdot \langle f, g\rangle = f \quad \wedge \quad \pi_2 \cdot \langle f, g\rangle = g \tag{19}$$

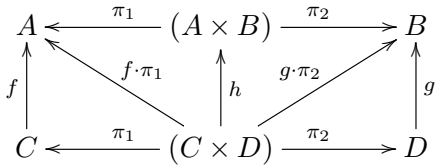
1.9.2. Produto de Funções

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:

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

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

vamos analisar o seguinte diagrama:



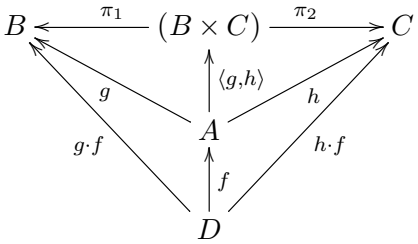
Podemos ver que $h = \langle f \cdot \pi_1, g \cdot \pi_2 \rangle$ está a mapear $C \times D$ para $A \times B$. Ora, isso corresponde à aplicação *paralela* de f e g e será expressa por $f \times g$. Assim, por definição:

$$f \times g \stackrel{\text{def}}{=} \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \tag{20}$$

1.9.3. Propriedades do Produto

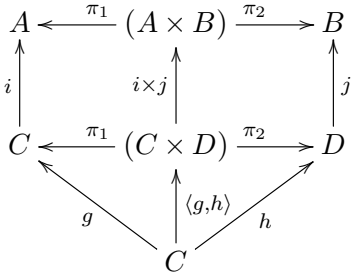
- A composição funcional e o combinador *split* relacionam-se um com o outro através da propriedade *fusão*-$\times$:

$$\langle g, h \rangle \cdot f = \langle g \cdot f, h \cdot f \rangle \tag{21}$$



- "O combinador *split* absorve o $\times$", como consequência da *fusão*-$\times$ e do *cancelamento*-$\times$[3] através da propriedade *absorção*-$\times$:

$$(i \times j) \cdot \langle g, h \rangle = \langle i \cdot g, j \cdot h \rangle \tag{22}$$



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

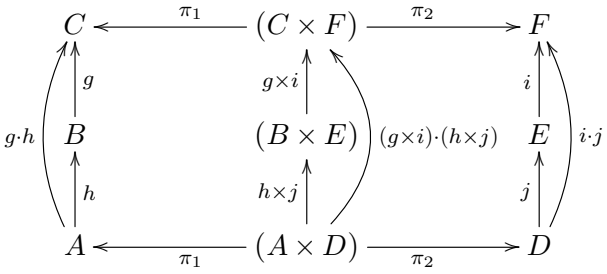
$$i \cdot \pi_1 = \pi_1 \cdot (i \times j) \tag{23}$$
$$j \cdot \pi_2 = \pi_2 \cdot (i \times j) \tag{24}$$

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*-$\times$:

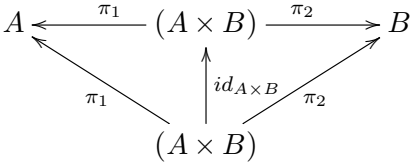
$$(g \cdot h) \times (i \cdot j) = (g \times i) \cdot (h \times j) \tag{25}$$

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



- 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-*$\times$:

$$\langle \pi_1, \pi_2 \rangle = id_{A \times B} \tag{26}$$



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

$$\frac{B \xleftarrow{f} A \qquad C \xleftarrow{g} A}{B \times C \xleftarrow{\langle f,g \rangle} A} \qquad \frac{C \xleftarrow{f} A \qquad D \xleftarrow{g} B}{C \times D \xleftarrow{f \times g} A \times B} \tag{27}$$

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

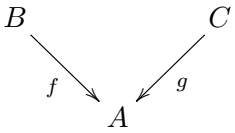
1.9.4. Combinador Either

Sejam f e g as seguintes funções:

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

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

Vamos "ligar" as duas setas, da uma forma semelhante à que 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 definimos o combinador "either f or g" e a notação a ser usada é $[\, f, g \,]$. Obviamente, o seu contradomínio 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[4]:

4 $B + C$ é chamado o *coproduto* de B e C.

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

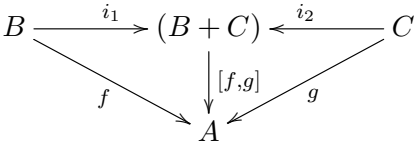
onde, em HASKELL, $i_1 = Left$ e $i_2 = Right$ cujos tipos são $B \xrightarrow{i_1} B + C \xleftarrow{i_2} C$ [5], e, no *Standard Prelude*, o tipo de dados $B + C$ é definido por:

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

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

$$\begin{array}{rcl} [\,f,g\,] & : & B + C \to A \\ [\,f,g\,]\ x & \stackrel{\text{def}}{=} & \begin{cases} x = i_1\ b \Rightarrow f\ b \\ x = i_2\ c \Rightarrow g\ c \end{cases} \end{array} \tag{28}$$

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



Nota: É interessante notar o quão semelhantes são os dois diagramas – 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 podemos dizer para o produto $A \times B$ pode ser transformado para o coproduto $A + B$.

5 As funções i_1 e i_2 são geralmente referidas como as *injeções* da união disjunta.

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 $f \times g$:

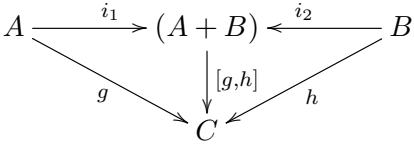
$$f + g \stackrel{\text{def}}{=} [i_1 \cdot f, i_2 \cdot g] \tag{29}$$

1.9.6. Propriedades do Coproduto

Podemos também definir algumas propriedades, assim como fizemos com o combinador produto:

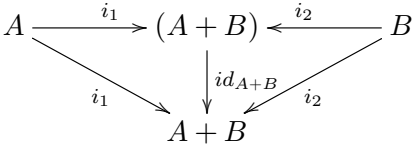
- *cancelamento-+*:

$$[g, h] \cdot i_1 = g, [g, h] \cdot i_2 = h \tag{30}$$



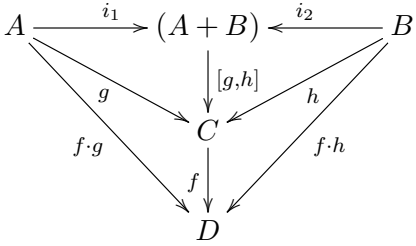
- *reflexão-+*:

$$[i_1, i_2] = id_{A+B} \tag{31}$$



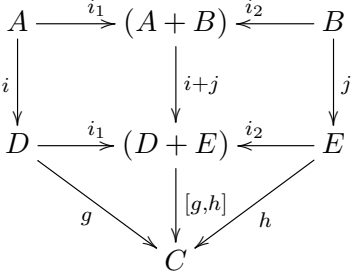
- *fusão-+*:

$$f \cdot [g, h] = [f \cdot g, f \cdot h] \tag{32}$$



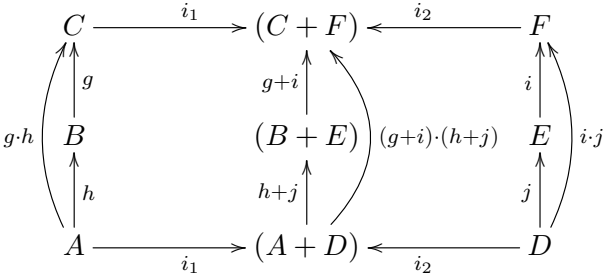
- *absorção-+*:

$$[g, h] \cdot (i + j) = [g \cdot i, h \cdot j] \tag{33}$$



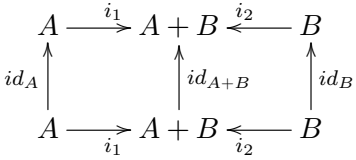
- *functor-+*:

$$(g \cdot h) + (i \cdot j) = (g + i) \cdot (h + j) \tag{34}$$



- *functor-id+*:

$$id_A + id_B = id_{A+B} \tag{35}$$



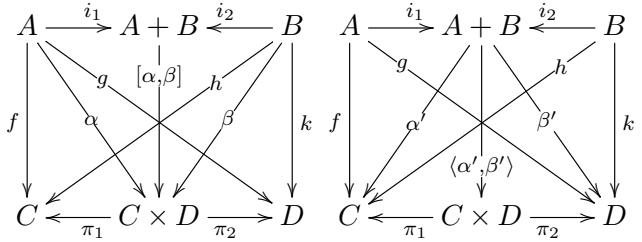
Resumindo, as regras, com respeito aos tipos, dos combinadores *either* e *soma* são:

$$\frac{C \xleftarrow{f} A \qquad C \xleftarrow{g} B}{C \xleftarrow{[f,g]} A + B} \qquad \frac{C \xleftarrow{f} A \qquad D \xleftarrow{g} B}{C + D \xleftarrow{f+g} A + B} \tag{36}$$

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*:

$$[\langle f, g\rangle, \langle h, k\rangle] = \langle [f, h], [g, k]\rangle \tag{37}$$



$$\begin{array}{lll} \sigma = [\alpha, \beta] & \sigma' = \langle \alpha', \beta' \rangle & \\ \alpha = \langle f, g \rangle & \alpha' = [f, h] & \boxed{\sigma = \sigma'} \\ \beta = \langle h, k \rangle & \beta' = [g, k] & \end{array}$$

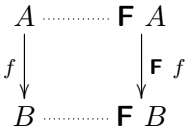
1.10 FUNCTORES

O conceito de functor, que nos é emprestado da teoria das categorias, 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.



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

- Functor-F:

$$\mathbf{F}\ (g \cdot h) = (\mathbf{F}\ g) \cdot (\mathbf{F}\ h) \tag{38}$$

- Functor-id-F:

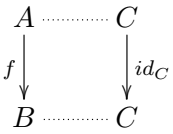
$$\mathbf{F}\ id_A = id_{(\mathbf{F}\ A)} \tag{39}$$

Dois exemplos básicos de functores são:

- Functor identidade:

$$\begin{aligned} \mathbf{F}\ X &= X \\ \mathbf{F}\ f &= f \end{aligned} \tag{40}$$

- Functor constante:



$$\begin{aligned} \mathbf{F}\ X &= C \\ \mathbf{F}\ f &= id_C \end{aligned} \tag{41}$$

1.10.1. Functores Polinomiais

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

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

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

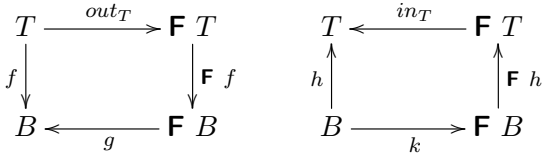
- o functor identidade ou um functor constante;
- o produto ou coproduto (soma) finito de functores polinomiais;
- a composição de functores polinomiais.

Dado o exemplo em cima, podemos derivar o seguinte:

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

Observemos que $1 + A \times X$ denota o mesmo que $id + id \times f$.

O 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:

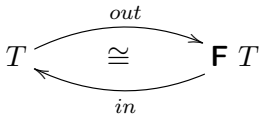


onde a função *out* "decompõe" o tipo de dados T numa determinada estrutura e a função *in* "constrói", a partir dessa mesma estrutura, o tipo de dados em causa.

Por fim, a função *in* pertence a uma classe própria de funções chamadas de F-*algebras*.

Uma F-*algebra* é qualquer função α com a assinatura $A \xleftarrow{\alpha} \mathsf{F}\ A$ 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 $\mathsf{F}\ T$ como o *functor tipo* associado a T.

Como exemplo, vamos recorrer ao conceito de lista aprendido em Programação Funcional: uma lista ou é vazia ou então é

composta por uma cabeça (primeiro elemento da lista) e uma cauda (a restante lista).[6] Assim, podemos definir o seguinte *isomorfismo* de construção de listas:

$$\mathit{List}\ A \underset{in}{\overset{out}{\cong}} 1 + A \times \mathit{List}\ A$$

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

Nota: 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:

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

1.10.2. Propriedades Naturais

A propriedade natural é geralmente chamada de *propriedade grátis*, uma vez que é facilmente derivada de qualquer fun-

6 **Nota**: No caso da lista singular, a cauda é a lista vazia.

ção polimórfica, isto é, cujo domínio e contradomínio podem assumir várias "formas" ou "padrões".

$$\begin{array}{ccc} A & \mathsf{F}\ A \xleftarrow{\phi} & \mathsf{G}\ A \\ {\scriptstyle f}\downarrow & {\scriptstyle \mathsf{F}\ f}\downarrow & \downarrow{\scriptstyle \mathsf{G}\ f} \\ B & \mathsf{F}\ B \xleftarrow[\phi]{} & \mathsf{G}\ B \end{array} \qquad (\mathsf{F}\ f) \cdot \phi = \phi \cdot (\mathsf{G}\ f) \quad (42)$$

Nota: 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:

$$\begin{array}{ccc} A^* & \xleftarrow{reverse} & A^* \\ {\scriptstyle map\ f}\downarrow & & \downarrow{\scriptstyle map\ f} \\ B^* & \xleftarrow[reverse]{} & B^* \end{array}$$

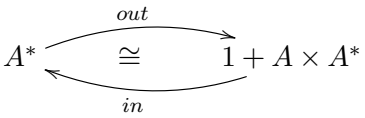
$$map\ f \cdot reverse = reverse \cdot map\ f$$

1.11 BI-FUNCTORES

Em 1.10 introduzimos a noção de *tipo indutivo* e do seu *functor tipo* associado, conforme representado no seguinte diagrama:

$$\mathsf{T} \underset{in}{\overset{out}{\cong}} \mathsf{F}\ \mathsf{T}$$

Vimos também o caso particular das listas não vazias.



Mas, existe um problema!

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

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

Não conseguimos definir, pois T não nos permite ter uma visão paramétrica de si mesmo. Surge então a necessidade de um functor binário que nos permita definir **F** T.

Supondo que apenas um parâmetro é identificado em T, definimos

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

onde **B** é o *bi-functor de base*[7] 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,

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

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

1.12 TIPOS INDUTIVOS RELEVANTES

T	in	$\mathbf{B}(X,Y)$	$\mathbf{B}(f,g)$	$\mathbf{F}\ f$
$\mathbb{N}_0$	[*0, succ*]	Não existe	Não existe	$id + f$
A^*	[*nil, cons*]	$1 + X \times Y$	$id + f \times g$	$\mathbf{B}(id, f)$
LTree A	[*Leaf, Fork*]	$X + Y^2$	$f + g^2$	$\mathbf{B}(id, f)$
BTree A	[*Empty, Node*]	$1 + X \times Y^2$	$id + f \times g^2$	$\mathbf{B}(id, f)$

Nota: 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, $\mathbf{F}\ X = 1 + X$.

Questão: o tipo de dados $\mathbb{N}_0$ não pode ser parametrizado? Como assim?

De uma forma muito informal (pedimos que o professor J.N. Oliveira nos desculpe pela ousadia), enquanto que podem existir listas e árvores de um qualquer tipo de dados, os naturais já são em si um tipo concreto de dados. Podemos definir listas e árvores de valores de um tipo qualquer A, mas não podemos definir naturais de um tipo qualquer A (por exemplo, não existem naturais de *booleanos*). Dito desta forma, até parece que não somos estudantes de engenharia!

2

CONDICIONAL DE MCCARTHY

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

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

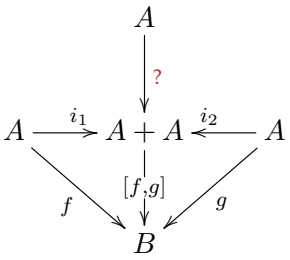
onde p é um predicado envolvendo x.

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 $[f, g]$. Uma vez que o objetivo é aplicar o predicado p, vamos relembrar os seguintes isomorfismos estudados em 13:

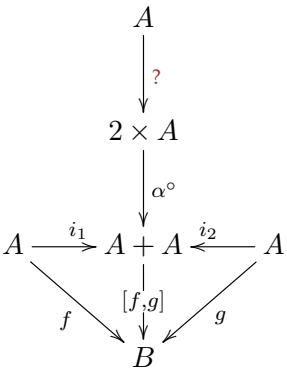
$$2 \cong \mathbb{B}$$

a partir do qual podemos inferir

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

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

É agora intuitiva a seguinte evolução no diagrama:

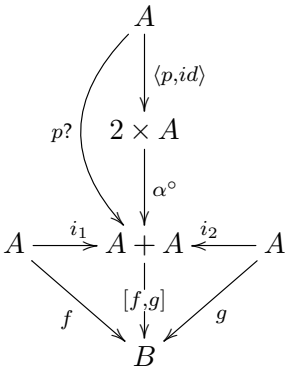


Falta apenas definir a função $2 \times A \xleftarrow{?} A$. Uma vez que a função retorna um produto, usar-se-á o combinador *split*. Por um lado, executa-se o predicado p dando origem a um *booleano* ($2 \cong \mathbb{B}$), e por outro, preserva-se o argumento com recurso à função identidade.

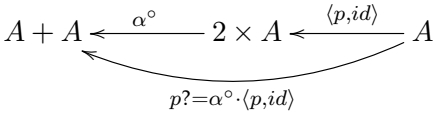
Obtém-se:

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

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 predicado 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 Depois do cientista da computação John McCarthy o definir.

2.1 PROPRIEDADES

- Natural-guarda:

$$p? \cdot f = (f + f) \cdot (p \cdot f)? \tag{43}$$

- 1ª Lei de fusão:

$$f \cdot (p \to g, h) = p \to f \cdot g, f \cdot h \tag{44}$$

- 2ª Lei de fusão:

$$(p \to f, g) \cdot h = (p \cdot h) \to (f \cdot h), (g \cdot h) \tag{45}$$

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

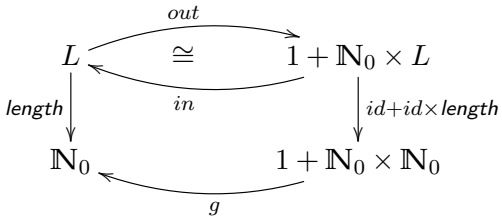
Vamos relembrar o *isomorfismo* de construção de listas estudado em 1.11. Assim, seja L o tipo que representa listas de naturais. Estamos então perante o seguinte isomorfismo:

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

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 0 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, $g_2(x, y) = \mathsf{succ}\ y$, que pode ser reescrita no nível *pointfree* como $g_2 = \mathsf{succ} \cdot \pi_2$.

Por fim, temos:

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

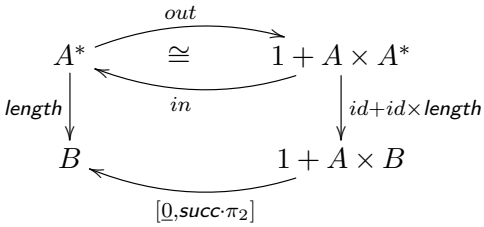
Diremos que g é o *gene* do catamorfismo *length* e usar-se-ão os *banana brackets* como notação. Assim,

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

Nota: Neste exemplo, o tipo L corresponde ao tipo $(\mathbb{N}_0)^*$ (listas de naturais).

* *é a notação usada para o tipo lista*

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:



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

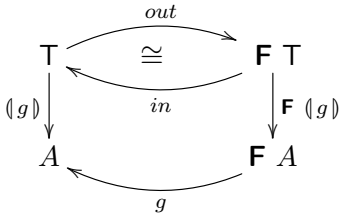
Obtivemos exatamente a mesma definição da função *length* apresentada em HASKELL no início da secção.

A última questão é: reconhecemos $(id + id \times \mathit{length})$ 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*:

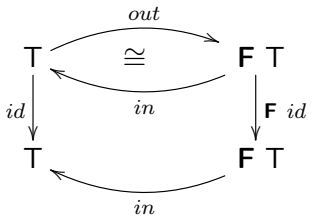
$$k = (\!|\, g \,|\!) \equiv k \cdot in = g \cdot \mathsf{F}\ k \tag{46}$$

- *cancelamento-cata*:

$$(\!|\, g \,|\!) \cdot in = g \cdot \mathsf{F}\ (\!|\, g \,|\!) \tag{47}$$

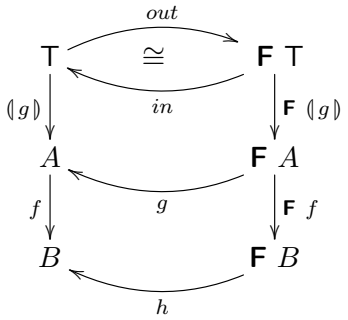
- *reflexão-cata*:

$$(\!|\, in \,|\!) = id_{\mathsf{T}} \tag{48}$$



- *fusão-cata*:

$$f \cdot (\!|\, g \,|\!) = (\!|\, h \,|\!) \Leftarrow f \cdot g = h \cdot \mathsf{F}\ f \tag{49}$$



- *absorção-cata*:

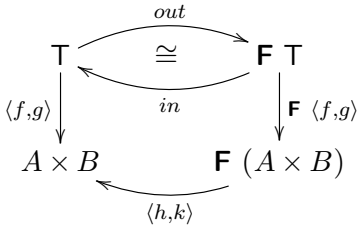
$$(\!|\, g \,|\!) \cdot \mathsf{T}\ f = (\!|\, g \cdot \mathsf{B}\ (f, id) \,|\!) \tag{50}$$

4

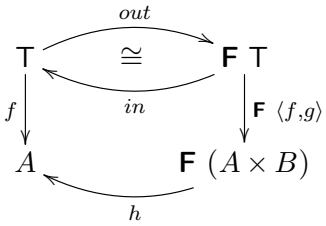
RECURSIVIDADE MÚTUA

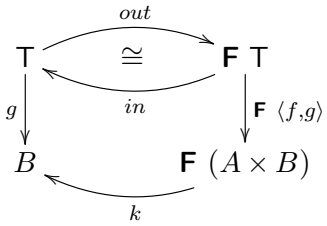
Este capítulo surge na resposta à pergunta: o que dizer sobre catamorfismos que geram pares como resultado?

A resposta é simples: basta recorrer ao combinador *split*. Olhemos então para 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:

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

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

$$
\begin{cases} f \cdot in = h \cdot \mathsf{F}\ \langle f, g\rangle \\ g \cdot in = k \cdot \mathsf{F}\ \langle f, g\rangle \end{cases} \equiv \langle f, g\rangle = (\!|\, \langle h, k\rangle \,|\!) \qquad (51)
$$

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:

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

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,

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

Similarmente,

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

Por fim,

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

$\equiv$ { lei de recursividade mútua }

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

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

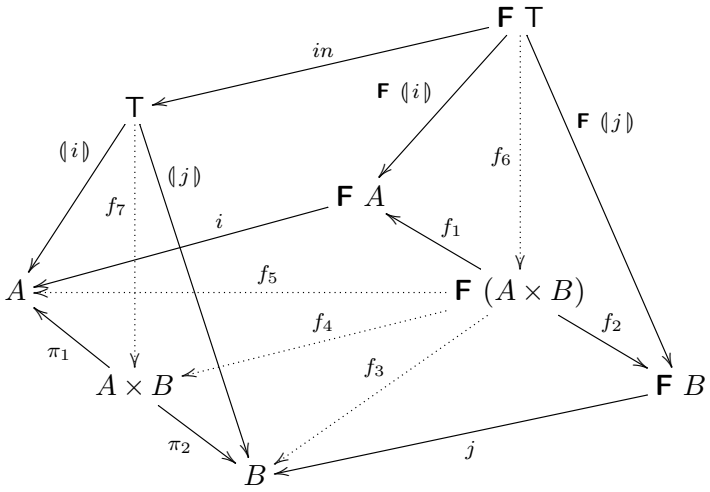
$$\langle (\!|\, i \,|\!), (\!|\, j \,|\!)\rangle = (\!|\, \langle i \cdot \mathsf{F}\ \pi_1, j \cdot \mathsf{F}\ \pi_2\rangle \,|\!)$$

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

$$\langle (\!|\, i \,|\!), (\!|\, j \,|\!)\rangle = (\!|\, (i \times j) \cdot \langle \mathsf{F}\ \pi_1, \mathsf{F}\ \pi_2\rangle \,|\!) \qquad (52)$$

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

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:

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

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, começando em 0 e 1, isto é,

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ \ldots$$

Podemos assim definir F_n do seguinte modo:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+1} &= F_n + F_{n-1},\ n > 1 \end{aligned}$$

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.

Nota: uma vez que estamos perante o functor dos naturais,

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

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

Parte-se de 51.

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

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

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

$\equiv$ { fusão-+; reflexão-+; eq-+; igualdade extensional; def-comp; def-split; def-$\times$ }

$$\begin{cases} \begin{cases} f\ 0 = a \\ f\ (n+1) = h_2\ (f\ n, g\ n) \end{cases} \\ \begin{cases} g\ 0 = b \\ g\ (n+1) = k_2\ (f\ n, g\ n) \end{cases} \end{cases} \equiv \langle f, g\rangle = (\!|\, \langle [\underline{a}, h_2], [\underline{b}, k_2]\rangle \,|\!) \tag{53}$$

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 53 obtemos:

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

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

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.

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

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

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

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.

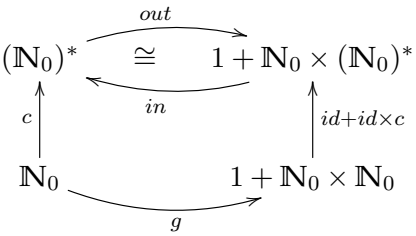
Tal como para os catamorfismos, vamos recorrer a um exemplo.

5.1 ANAMORFISMO SOBRE LISTAS

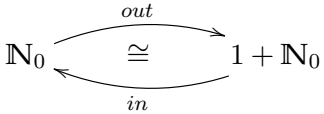
Como exemplo, vamos considerar a função c que, dado um $n \in \mathbb{N}$, constrói a lista $[1..n]$ invertida (ex.: $c\ 4 = [4, 3, 2, 1]$). Pode ser escrita em HASKELL da seguinte forma:

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

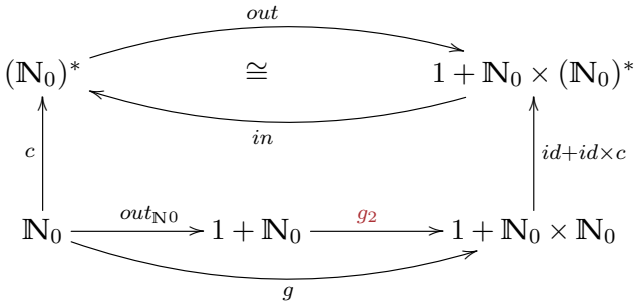
Vamos começar por analisar o seguinte diagrama e tentar definir a função g:



Relembremos o isomorfismo de construção de naturais dado em 17.



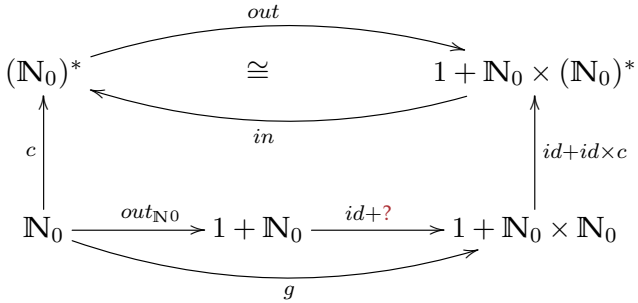
A função *out* é muito útil para a definição de g, pelo que a vamos inserir no diagrama com o nome $out_{\mathbb{N}0}$ para não confundir com a função *out* para listas já presente no diagrama.



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.

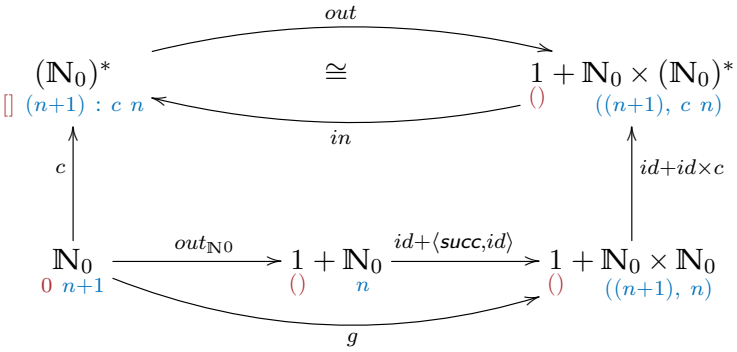
Quando g recebe um valor $n+1$, a função $out_{\mathbb{N}0}$ injeta o valor n à direita.[1] O objetivo é aplicar recursivamente a função c ao valor n, e, no final, construir a lista $((n+1)\ :\ c\ n)$. Ora, tendo o valor n, basta apenas formar o par $(n+1, n)$, visto que na segunda seta vertical, $n+1$ 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 $g = (id + \langle \mathsf{succ}, id\rangle) \cdot out_{\mathbb{N}0}$ e diremos que g é o *gene* do anamorfismo c, usando a seguinte notação:

$$c = [\![(g)]\!] = [\![((id + \langle \mathsf{succ}, id\rangle) \cdot out_{\mathbb{N}0})]\!]$$

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 $g\ (n+1)$.



$$c = in_L \cdot (id + id \times c) \cdot (id + \langle succ, id \rangle) \cdot out_{\mathbb{N}0}$$

$\equiv$ { def-in_L; isomorfismo *in* e *out* }

$$c \cdot in_{\mathbb{N}0} = [nil, cons] \cdot (id + id \times c) \cdot (id + \langle succ, id \rangle)$$

$\equiv$ { def-$in_{\mathbb{N}0}$; absorção-+ 2x }

$$c \cdot [\underline{0}, succ] = [nil, cons \cdot (id \times c) \cdot \langle succ, id \rangle]$$

$\equiv$ { fusão-+; eq-+; absorção-×; natural-*id* }

$$\begin{cases} c \cdot \underline{0} = nil \\ c \cdot succ = cons \cdot \langle succ, c \rangle \end{cases}$$

$\equiv$ { igualdade extensional; def-comp; def-*split*; def-*succ* }

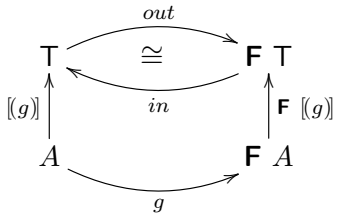
$$\begin{cases} c\ 0 = [\,] \\ c\ (n+1) = (n+1)\ :\ c\ n \end{cases}$$

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:

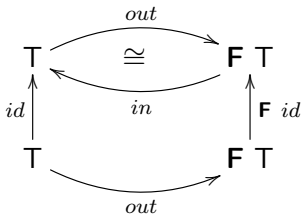
$$k = [\![(g)]\!] \equiv out \cdot k = (\mathsf{F}\ k) \cdot g \tag{54}$$

- Cancelamento-ana:

$$out \cdot [\![(g)]\!] = \mathsf{F}\ [\![(g)]\!] \cdot g \tag{55}$$

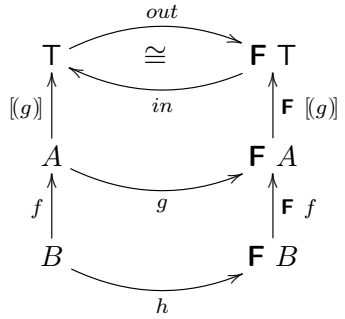
- Reflexão-ana:

$$[\!(out)\!] = id_{\mathsf{T}} \tag{56}$$



- Fusão-ana:

$$[\!(g)\!] \cdot f = [\!(h)\!] \Leftarrow g \cdot f = \mathsf{F}\ f \cdot h \tag{57}$$



- Absorção-ana:

$$\mathsf{T}\ f \cdot [\!(g)\!] = [\!(\mathsf{B}(f, id) \cdot g)\!] \tag{58}$$

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:

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

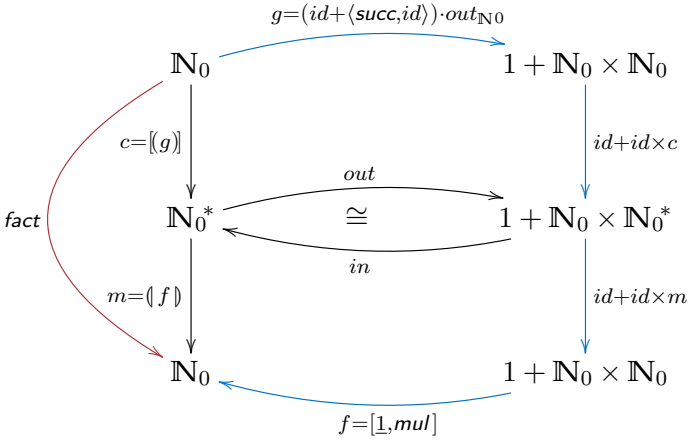
Então, é necessário criar uma estrutura que contenha todos os valores a multiplicar, e depois consumir a estrutura, multiplicando todos os valores nela presentes.

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:

$$\mathit{fact} = m \cdot c$$

e pode ser facilmente interpretada a partir do seguinte diagrama:



A notação a ser usada é a seguinte:

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

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

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

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

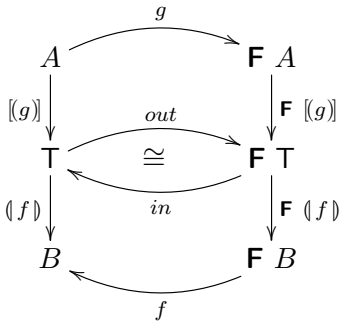
Por fim, fazendo uma última referência à abrangente e completa clássica língua grega, *hilomorfismo* deriva de: *hylo* "matéria, coisa, tudo" + *morphe* "forma". Isto sugere que o *hilomorfismo* é um esquema geral, ou seja, qualquer programa que possamos escrever é uma composição de um *catamorfismo* com um *anamorfismo*.

Nota: 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

$$[\!|\, f, g \,|\!] = (\!|\, f \,|\!) \cdot [\!(g)\!] \tag{59}$$



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*.

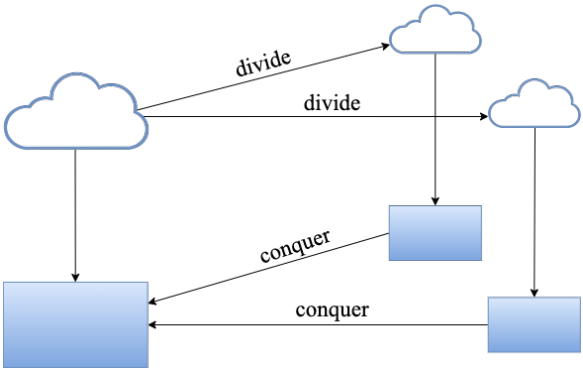


Figura 1: Ilustração da estratégia *Divide & Conquer*

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.

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

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*.

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

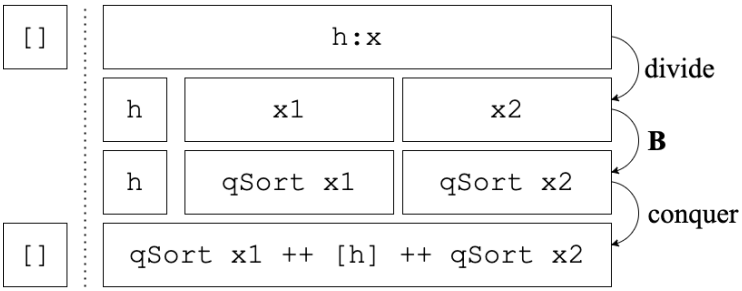
Vamos ver um exemplo.

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
    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:



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:

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

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

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

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

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

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

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

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

$$\equiv \quad \{ \text{ isomorfismo } \mathit{in} \text{ e } \mathit{out} \text{ } \}$$

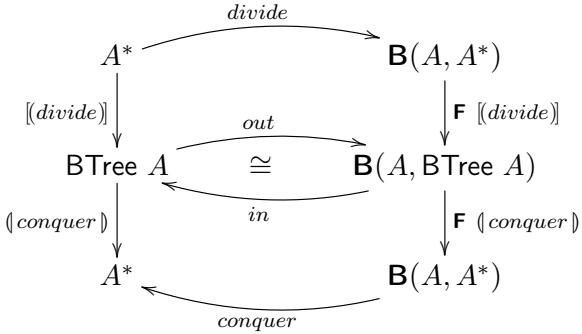
$$qSort = \underbrace{[nil, f]}_{conquer} \cdot \underbrace{(id + id \times qSort^2)}_{\mathbf{B}(id, qSort)} \cdot \underbrace{(id + g) \cdot out}_{divide}$$

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.

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

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]

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.

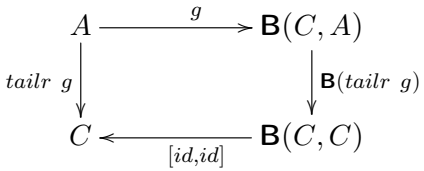


Por fim,

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

6.3 TAIL RECURSION

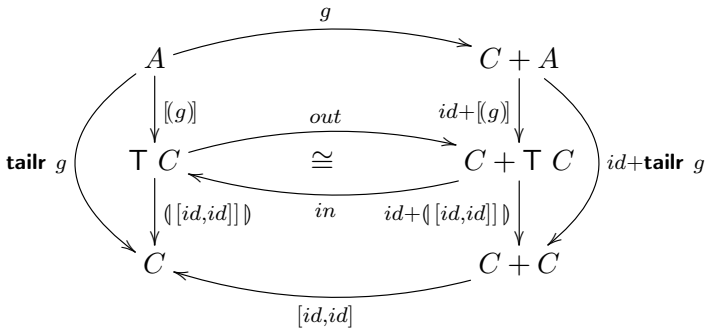
Como já referido, este hilomorfismo é um caso particular cujo anamorfismo (*divide*) praticamente produz a solução. O catamorfismo (*conquer*) apenas "retira" a solução (já produzida) da estrutura intermédia.



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

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

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



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

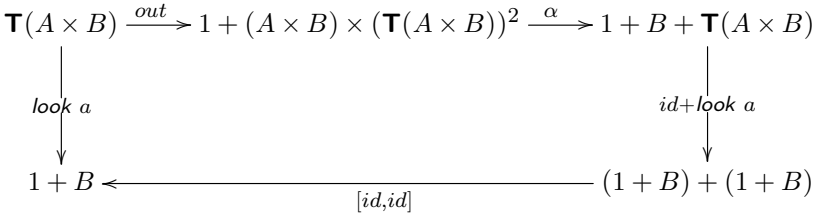
Podemos então olhar para T C como um tipo semelhante ao tipo dos números naturais, com a diferença de que guarda informação no 0. Assim, T C, além de conter a solução C, contém também o número de iterações feitas.[3]

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

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

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'
        | 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 uma forma informal, podemos definir um *mónade* como um functor com propriedades especiais. Mas, para entender a importância destes functores "especiais", temos de aprofundar o nosso estudo sobre funções parciais. Só assim poderemos entender o que é um *mónade*.

7.1 FUNÇÕES PARCIAIS

O que é que acontece quando "interrogamos" o GHCi com

```
> head []
```

?

Ocorre um erro. Porquê? Porque a lista vazia não tem nenhum elemento, por isso não há nenhum valor que seja possível retornar. 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.

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

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

Ora, para compor uma função, digamos k, com g, é necessário que k tenha como domínio $1 + B$.

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

Questão: e se a função k não tem como domínio $1 + B$, mas apenas B?

Questão: é possível compor funções parciais?

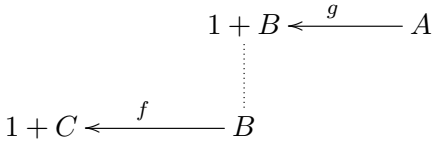
7.2 COMPOSIÇÃO DE FUNÇÕES PARCIAIS

Sejam f e g as seguintes funções:

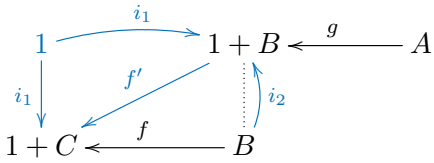
$$1 + C \xleftarrow{f} B$$

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

Como é que vamos compor as duas funções?



Não podemos simplesmente fazer $f \cdot g$, visto que f requer um valor-B e a função g pode retornar $()$. Vamos ter de utilizar uma função que está "escondida" no diagrama.



(60)

Uma vez que $f' = [i_1, f]$, podemos assim definir a composição de funções parciais do seguinte modo:

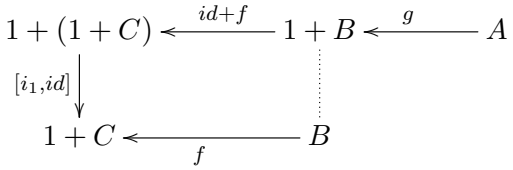
$$f \bullet g \stackrel{\text{def}}{=} [i_1, f] \cdot g \tag{61}$$

7.3 COMPOSIÇÃO DE KLEISLI

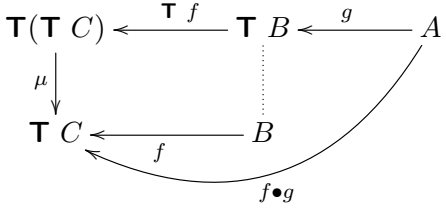
A partir do diagrama 60, podemos inferir a definição de f' como sendo:

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

e definir o seguinte diagrama:



Reparemos que $(id + f)$ é o functor em causa! Assim, podemos generalizar o diagrama da *composição de Kleisli*:



Notamos algo desconhecido no diagrama? Sim! A função μ! Bem, esta função é uma das razões para um *mónade* ser chamado de um "functor de corrida". Estamos agora prontos para entender 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:

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

e obedece às duas seguintes propriedades:

$$\mu \cdot u = \mu \cdot \mathsf{T}\ u = id \tag{62}$$

$$\mu \cdot \mu = \mu \cdot \mathsf{T}\ \mu \tag{63}$$

7.4.1. Exemplo: Mónade LTree

Uma *Leaf Tree* é definida em Haskell do seguinte modo:

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

É muito intuitivo definir o *isomorfismo* de construção de *Leaf Trees*:

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

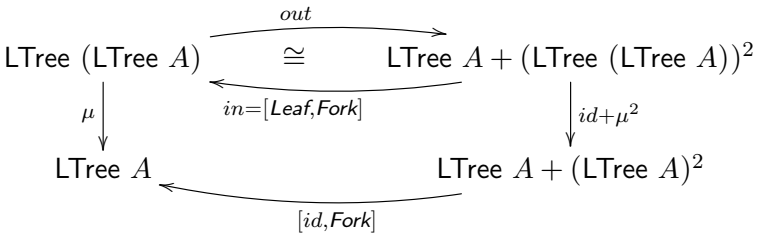
Como estudamos, um mónade dispõe de duas funções, u e μ.

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

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*.



Concluímos que $\mu = (\!|\, [id, \mathit{Fork}] \,|\!)$. Por fim, é necessário verificar as leis 62 (unidade) e 63 (multiplicação).

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

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

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

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

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

Assim,

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

□

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

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

Concluímos assim que *LTree A* é um mónade.

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

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

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

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

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

$$u = in \cdot i_1$$

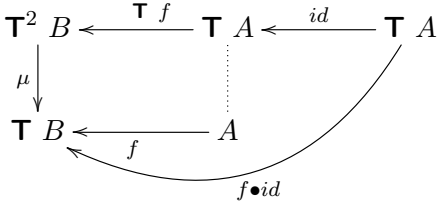
7.5 PROPRIEDADES NATURAIS

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

$$\begin{cases} \mathsf{T}\ f \cdot u = u \cdot f \\ \mathsf{T}\ f \cdot \mu = \mu \cdot \mathsf{T}^2\ f \end{cases} \tag{64}$$

7.6 APLICAÇÃO MONÁDICA – BINDING

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

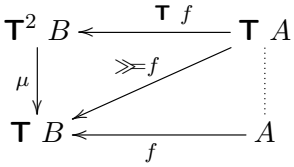


Concluímos que:

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

A notação a ser usada é a seguinte:

$$x \gg\!= f \stackrel{\text{def}}{=} (\mu \cdot \mathsf{T}\ f)\ x \tag{65}$$



7.7 NOTAÇÃO-DO

$$(f \bullet g)\ a = \mathsf{do}\ \{\ b \leftarrow g\ a\ ;\ f\ b\ \} \tag{66}$$
$$(f \bullet id)\ x = x \gg\!= f = \mathsf{do}\ \{\ b \leftarrow x\ ;\ f\ b\ \} \tag{67}$$

7.8 SISTEMAS REATIVOS

Esta subsecção ainda não está disponível.

7.8.1. Monad Estado

Esta subsecção ainda não está disponível.

7.9 RECURSIVIDADE MONÁDICA

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

A

CÁLCULO DE QUANTIFICADORES DE EINDHOVEN

A.1 NOTAÇÃO

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

A.2 REGRAS

- Trading:

$$\langle \forall k : R \wedge S : T \rangle = \langle \forall k : R : S \Rightarrow T \rangle \quad (68)$$

$$\langle \exists k : R \wedge S : T \rangle = \langle \exists k : R : S \wedge T \rangle \quad (69)$$

- de Morgan:

$$\neg\langle\forall k : R : T\rangle = \langle\exists k : R : \neg T\rangle \tag{70}$$

$$\neg\langle\exists k : R : T\rangle = \langle\forall k : R : \neg T\rangle \tag{71}$$

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.