

Cálculo de Programas

Temas

- Introdução ao Cálculo de Programas
 - Teoria Algébrica da Programação
 - Aplicação: Especificação e Transformação de Programas
- Programação Funcional “avançada”: Padrões de Recursividade
- Programação Genérica

Avaliação

- Componente Prática **obrigatória** - 30% (min. 8.5)

Datas (submissão electrónica):

10/10

7/11

12/12

- Componente Teórica: exame - 70% (min. 8.5)

Equipa Docente

- Jorge Sousa Pinto (T)
- Olga Pachedo (TP, P)
- Luís Soares Barbosa (TP, P)

{jsp,omp,lsb}@di.uminho.pt

Bibliografia

- *J.N. Oliveira:*
 - “*An Introduction to Pointfree Programming*”
 - “*Recursion in the Pointfree Style*”
- *Bird, R., de Moor, O.:*
 - “*Algebra of Programming*”, Prentice Hall, 1997

Mais Informação

- Página da disciplina acessível em:

<http://www.di.uminho.pt/~jsp>

I - Revisão

Conceitos Básicos - Modelos de Programação

1- O que é um programa?

2- O que é um algoritmo?

- Método bem definido para a resolução de um problema
- Haverá algo de especificamente “informático” neste conceito?

3- O que é um programa em código-máquina?

- Arquitectura de “von Neumann”
- E um programa escrito em *assembly*?

4- O que é uma linguagem de “alto nível”?

- Riqueza expressiva, operações poderosas
- Tradução para o nível da máquina
- Tipos Complexos

5- O que é um tipo?

- E para que serve?
- *type-checking*
- ajuda para o programador

1- O que é um programa?

Uma resposta possível - uma entidade que:

- obedece a uma SINTAXE conhecida
(linguagem de programação)
- destina-se a ser traduzido em código-máquina de um computador moderno

1- O que é um programa?

- a sua execução em computador corresponde a uma concretização de um algoritmo para a resolução de um problema
- o seu comportamento (execução) segue o especificado pela SEMÂNTICA da linguagem de programação (formal / informal)

6- O que é uma linguagem imperativa?

- Modelo semelhante ao da máquina
- Sequenciação de instruções
- Mas mais rico nas estruturas de controlo, tipos de dados, modularidade, etc.
- Alocação de memória estática vs. dinâmica

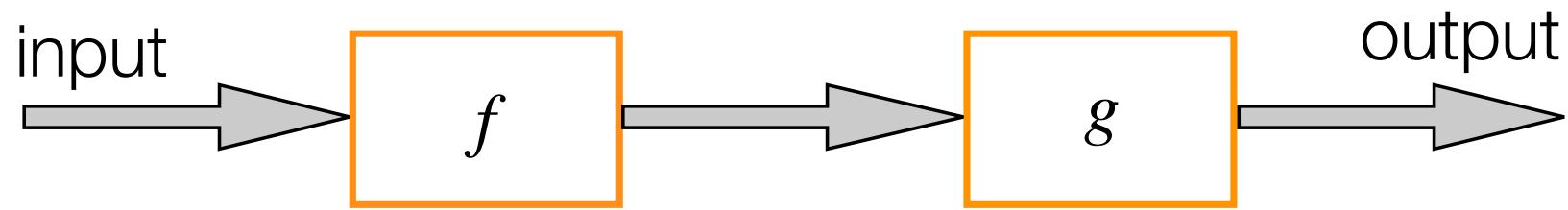
7- O que é um programa funcional?

- E em que se distingue de um imperativo?
- Linguagem baseada num *modelo matemático*
- Entidades básicas são *funções* e não *instruções*
- Sequenciação de instruções vs
Composição de funções

8- O que é o estado de um programa imperativo?

- Conjunto de estruturas de dados alocadas estáticamente ou dinamicamente pelo programa
- Que tipos de instruções podem *modificar* o estado de um programa?
- Modelo imperativo - numa sequência $i_1 ; i_2$ instrução i_2 é executada no estado alterado por i_1

9- Existe estado nos programas funcionais?



Composição $g . f$

output de f é passado a g como input

9- Existe estado nos programas funcionais?

- A menos de *efeitos monádicos*, funções não têm acesso a um estado global

- Mas então o que significa

```
let x = 2
in (let x = 3
    in x + 4)
+ x
```

let é uma instrução de atribuição?

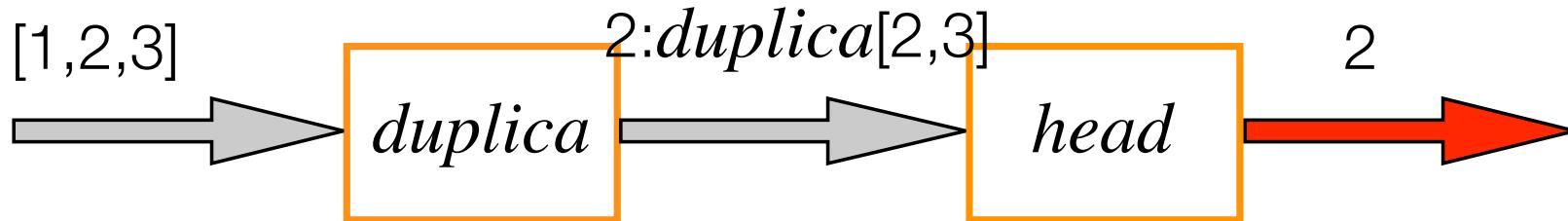
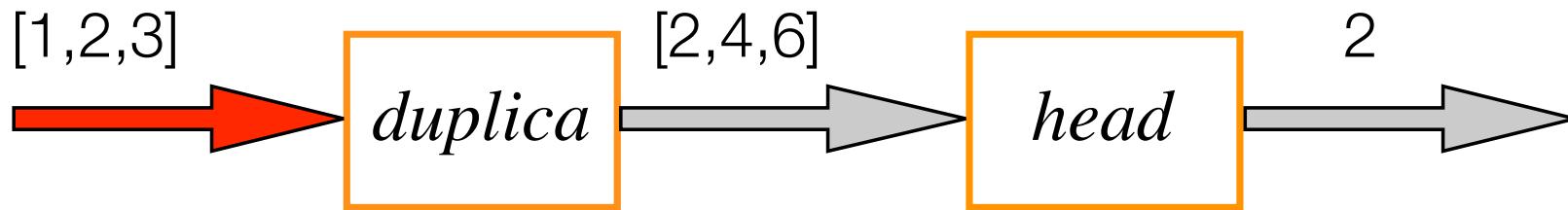
Exemplo de um PF

```
duplica [] = []
duplica (h:t) = (2*h):(duplica t)
```

- O que acontece à lista original?
- Que tipo de utilização de memória faz um programa funcional?
- O que é um *garbage-collector*?

Execução estrita vs. lazy

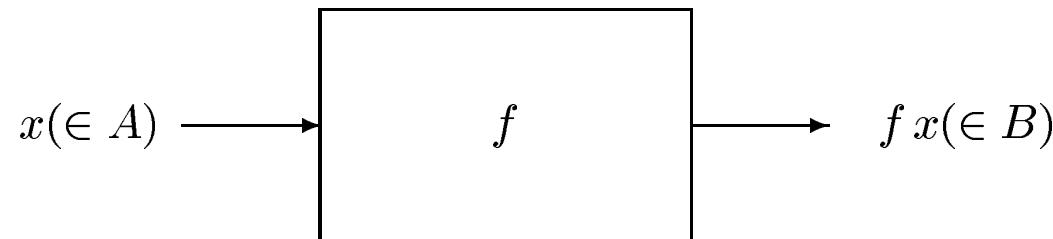
head(duplica [1,2,3])



II - Revisões

Funções, Diagramas, e Isomorfismos

1- Assinaturas



$$f : B \leftarrow A , f : A \longrightarrow B , B \xleftarrow{f} A \text{ or } A \xrightarrow{f} B$$

Em Haskell,

`f :: A -> B`

2- Aplicação

$f :: A \rightarrow B, x :: A$

$f\ x :: B$

$g :: A \rightarrow B \rightarrow C, x :: A, y :: B$

$g\ x :: B \rightarrow C$

$(g\ x)\ y :: C$

$g\ x\ y :: C$

Como associam a aplicação e o construtor \rightarrow ?

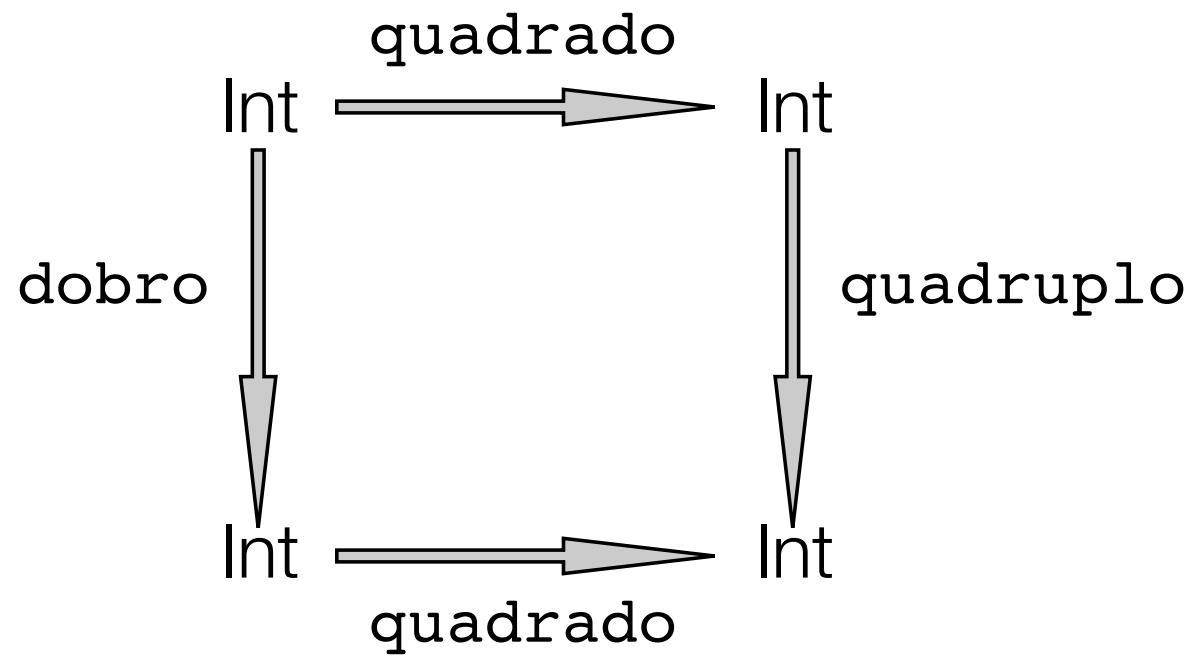
3- Igualdade, Diagramas e Composição

```
quadruplo x = 4 * x  
dobro  x = 2 * x  
quadrado x = x * x
```

(i) *quadruplo (quadrado x) = quadrado (dobro x)*

igualdade de valores (do mesmo tipo)

(ii)



comutação de um diagrama

(iii) $\text{quadruplo} \cdot \text{quadrado} = \text{quadrado} \cdot \text{dobro}$

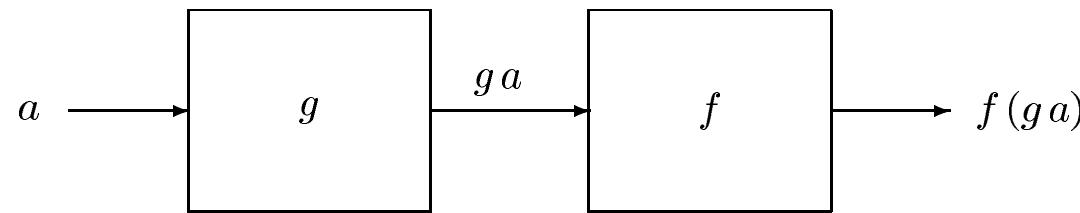
Igualdade de Funções

No cálculo de programas recorreremos frequentemente a diagramas comutativos e à noção de igualdade entre funções.

Mas que significado poderá ter esta igualdade?

4- Composição

$$(f \bullet g) a \stackrel{\text{def}}{=} f(g a)$$



Forma fundamental de *combinação* de funções

Relembrar: que condição se deve verificar nos tipos?

Diagrama:

$$\begin{array}{ccc} B & \xleftarrow{g} & A \\ f \downarrow & & \swarrow f \circ g \\ C & & \end{array}$$

Propriedade: *Associatividade*

$$(f \circ g) \circ h = f \circ (g \circ h)$$

5- Funções Identidade

$$\begin{array}{c} id_A : A \leftarrow A \\ id_A a \stackrel{\text{def}}{=} a \end{array}$$

- Trata-se de uma *família* de funções, indexada pelo tipo
- notação: omitiremos tipo quando não haja ambiguidade

Diagrama:

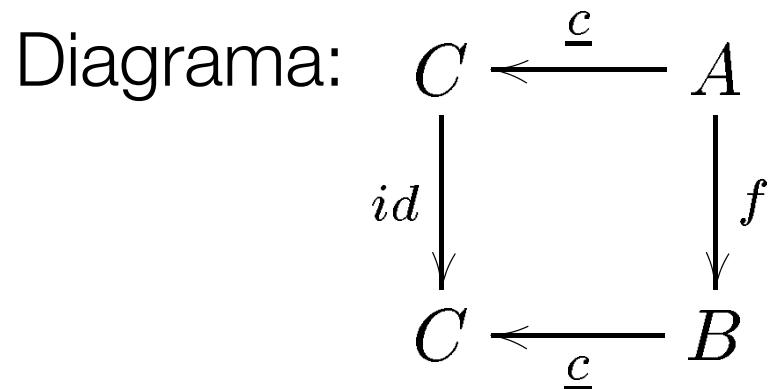
$$\begin{array}{ccc} A & \xleftarrow{id} & A \\ f \downarrow & & \downarrow f \\ B & \xleftarrow{id} & B \end{array}$$

Propriedade: *Elem. neutro composição*

$$f \bullet id = id \bullet f = f$$

6- Funções-constante

$$\begin{array}{ccc} \underline{c} & : & A \longrightarrow C \\ \underline{c}a & \stackrel{\text{def}}{=} & c \end{array}$$



Propriedade: *Fusão de funções-constante*

$$\underline{\mathcal{C}}_B \bullet f = \underline{\mathcal{C}}_A$$

$$\underline{\mathcal{C}} \bullet f = \underline{\mathcal{C}}$$

7- Isomorfismos

Um isomorfismo é:

- Uma função simultaneamente *injectiva* e *sobrejectiva*
- uma função *invertível*

$$B \xleftarrow{f} A \qquad B \xrightarrow{f^{-1}} A$$

$$f \cdot f^{-1} = id_B \quad \wedge \quad f^{-1} \cdot f = id_A$$

- Se existe um isomorfismo entre dois tipos, estes tipos dizem-se *isomorfos* e são “essencialmente” o mesmo tipo

$$A \cong B$$

- isomorfismo permite converter dados entre os dois tipos *sem perda de informação*

Exemplo

Weekday =

$\{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday\}$

$f : \text{Weekday} \longrightarrow \{1, 2, 3, 4, 5, 6, 7\}$

$f Monday = 1$

$f Tuesday = 2$

$f Wednesday = 3$

$f Thursday = 4$

$f Friday = 5$

$f Saturday = 6$

$f Sunday = 7$

Quantos isomorfismos haverá entre estes dois tipos?

Será a composição de dois isomorfismos necessariamente um isomorfismo?

III - Tipos-produto e split de funções

Como combinar duas funções que partilham o mesmo domínio?

1- Tipos-produto

- Produto cartesiano de dois tipos:

$$A \times B \stackrel{\text{def}}{=} \{(a, b) \mid a \in A \wedge b \in B\}$$

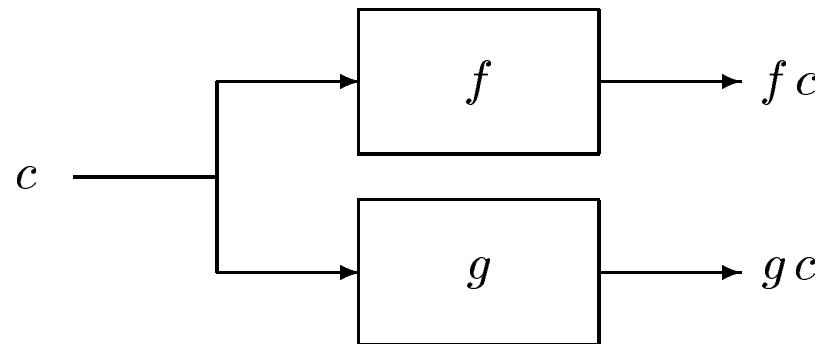
- Produto dos tipos A e B preserva informação de ambos

- Informação de um dos tipos pode ser recuperada através das funções de *projecção*:
(`fst` e `snd` em Haskell)

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

$$\pi_1(a, b) = a \quad \text{and} \quad \pi_2(a, b) = b$$

2- O Combinador Split



Ideia: aplicar simultaneamente duas funções com o mesmo domínio C a um ponto de C , obtendo-se assim um par de resultados

2- O Combinador Split

$$\begin{aligned}\langle f, g \rangle & : C \longrightarrow A \times B \\ \langle f, g \rangle c & \stackrel{\text{def}}{=} (f c, g c)\end{aligned}$$

$$\begin{array}{ccc} A & \xleftarrow{f} & C \\ & \xleftarrow{g} & \end{array}$$

Ideia: aplicar simultaneamente duas funções com o mesmo domínio C a um ponto de C, obtendo-se assim um par de resultados

3- Prop: Cancelamento

$$\pi_1 \cdot \langle f, g \rangle = f \quad \text{and} \quad \pi_2 \cdot \langle f, g \rangle = g$$

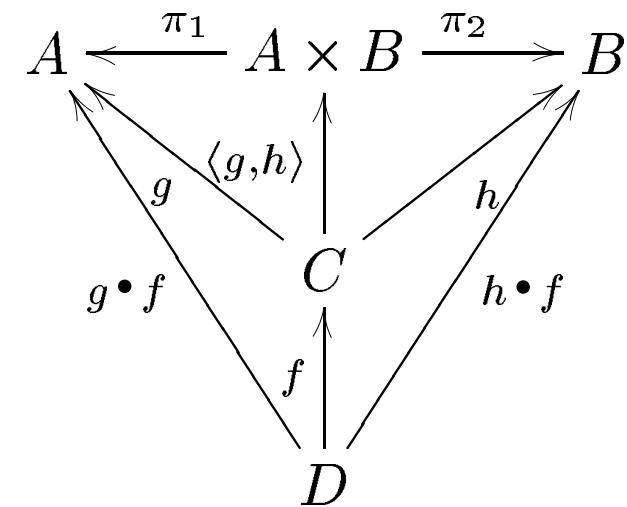
$$\begin{array}{ccccc} & & A \times B & & \\ & \swarrow & \downarrow \langle f, g \rangle & \searrow & \\ A & \xleftarrow{\pi_1} & & \xrightarrow{\pi_2} & B \\ & f & & g & \\ & C & & & \end{array}$$

comuta

Fácil de justificar com base nas definições *pointwise*

4- Propr: Fusão

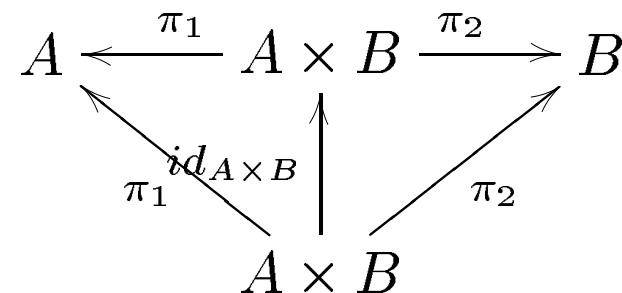
$$\langle g, h \rangle \bullet f = \langle g \bullet f, h \bullet f \rangle$$



Fácil de justificar com base nas definições *pointwise*

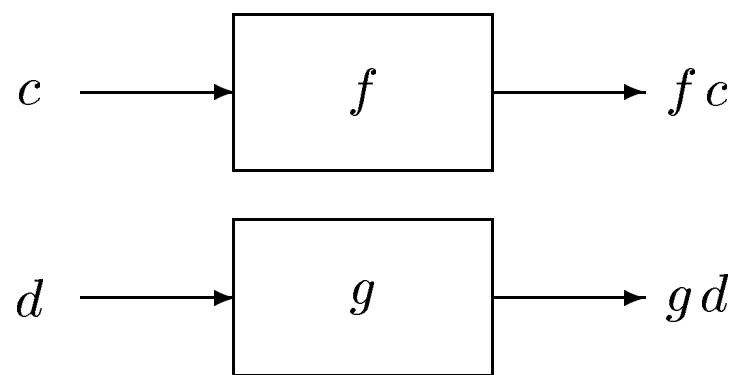
5- Propr: Reflexão

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



6- O Combinador Produto

- Os tipos-produto sugerem também uma forma de combinar quaisquer duas funções, mesmo quando os domínios não coincidem:



6- O Combinador Produto

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

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

$$\begin{array}{c} A \xleftarrow{f} C \\ B \xleftarrow{g} D \end{array}$$

Ideia: aplicar f e g em paralelo a um par de argumentos, obtendo-se um par de resultados

$$\text{Pointwise , } (f < g)(x, y) = (f x, g y)$$

7- Prop: Absorção

$$(i \times j) \bullet \langle g, h \rangle = \langle i \bullet g, j \bullet h \rangle$$

$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \\ i \uparrow & & i \times j \uparrow & & j \uparrow \\ D & \xleftarrow{\pi_1} & D \times E & \xrightarrow{\pi_2} & E \\ g \swarrow & \langle g, h \rangle \downarrow & & \nearrow h & \end{array}$$

Exercício: demonstrar esta propriedade

8- Proprs. Functoriais e outras

$$id_A \times id_B = id_{A \times B}$$

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

$$i \bullet \pi_1 = \pi_1 \bullet (i \times j)$$

$$j \bullet \pi_2 = \pi_2 \bullet (i \times j)$$

Exercícios...

IV - Tipos-coproduto e *either* de funções

Como combinar duas funções que partilham o mesmo codomínio?

1- Tipos-coproduto

- União disjunta ou *coproduto* de dois tipos:

$$A + B = \{(t_1, a) \mid a \in A\} \cup \{(t_2, b) \mid b \in B\}$$

(t_1, t_2 são *etiquetas distintas*)

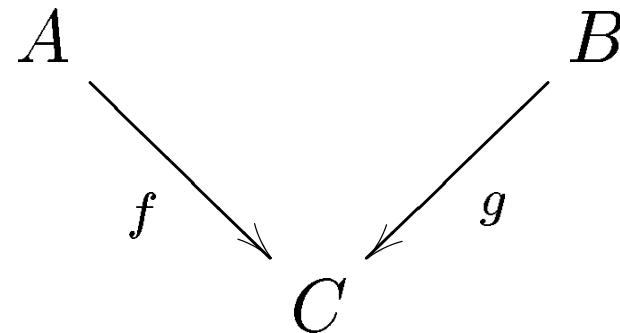
- Preserva informação de apenas um dos tipos

- Informação de um dos tipos é introduzida no tipo-coproduto por uma função de *injecção*:
(Left e Right em Haskell)

$$A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$$

$$i_1 a = (t_1, a) \quad , \quad i_2 b = (t_2, b)$$

2- O Combinador Either



Ideia: dadas duas funções com o mesmo codomínio, aplicar uma delas, de acordo com o tipo do argumento

2- O Combinador Either

$$[f, g] : A + B \longrightarrow C$$

$$[f, g]x \stackrel{\text{def}}{=} \begin{cases} x = i_1 a \Rightarrow fa \\ x = i_2 b \Rightarrow gb \end{cases}$$

$$\begin{array}{rcl} f & : & C \longleftarrow A \\ g & : & C \longleftarrow B \end{array}$$

Ideia: dadas duas funções com o mesmo codomínio, aplicar uma delas, de acordo com o tipo do argumento

3- Prop: Cancelamento

$$[g, h] \cdot i_1 = g, [g, h] \cdot i_2 = h$$

$$\begin{array}{ccccc} A & \xrightarrow{i_1} & A + B & \xleftarrow{i_2} & B \\ & \searrow g & \downarrow [g, h] & \swarrow h & \\ & & C & & \end{array}$$

comuta

Fácil de justificar com base nas definições *pointwise*

4- Propr: Fusão

$$\begin{array}{ccccc} A & \xrightarrow{i_1} & A + B & \xleftarrow{i_2} & B \\ & \searrow g & \downarrow [g,h] & \swarrow h & \\ & f \bullet g & C & f \bullet h & \\ & \downarrow f & & \downarrow & \\ & & D & & \end{array}$$

$f \bullet [g, h] = [f \bullet g, f \bullet h]$

Fácil de justificar com base nas definições *pointwise*

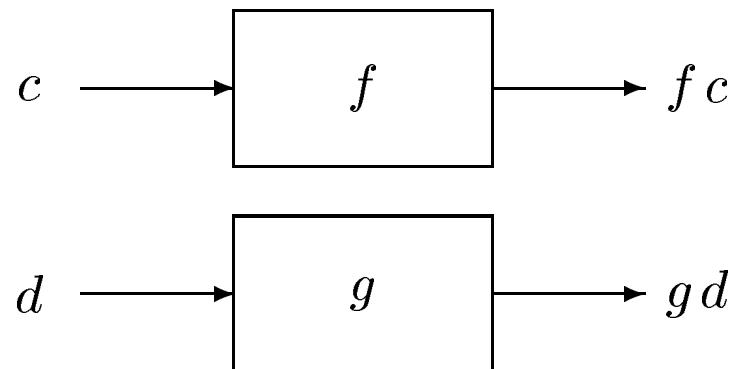
5- Propr: Reflexão

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

$$\begin{array}{ccccc} A & \xrightarrow{i_1} & A + B & \xleftarrow{i_2} & B \\ & \searrow i_1 & \downarrow id_{A+B} & \swarrow i_2 & \\ & & A + B & & \end{array}$$

6- O Combinador Soma

- Os tipos-coproduto sugerem também uma forma alternativa de combinar quaisquer duas funções, mesmo quando os codomínios não coincidem:



6- O Combinador Soma

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

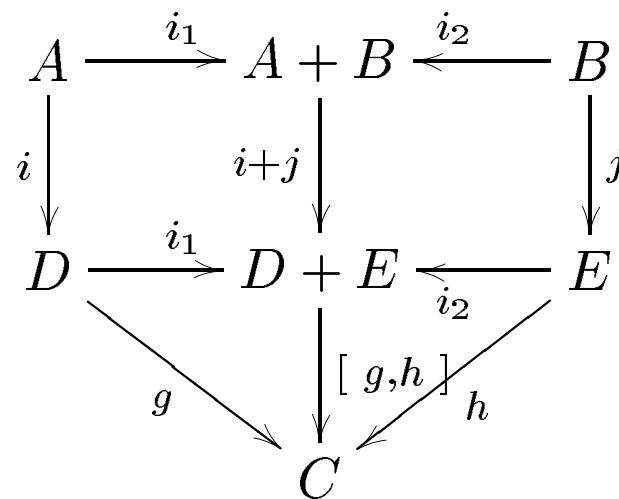
$$f + g \stackrel{\text{def}}{=} [i_1 \bullet f, i_2 \bullet g] \quad \begin{array}{c} A \xleftarrow{f} C \\ B \xleftarrow{g} D \end{array}$$

Ideia: aplicar f ou g a um argumento de tipo C ou D, obtendo-se um resultado de tipo A ou B

$$\text{Pointwise , } (f + g)(i1\ x) = i1(f\ x)$$

7- Prop: Absorção

$$[g, h] \bullet (i + j) = [g \bullet i, h \bullet j]$$



Exercício: demonstrar esta propriedade

8- Proprs. Functoriais

$$id_A + id_B = id_{A+B}$$

$$(g \bullet h) + (i \bullet j) = (g + i) \bullet (h + j)$$

Exercícios...

9- Coprodutos em Haskell

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

Left e Right desempenham o papel
de *injecções*

V - Produtos e coprodutos

Notas e Exemplos

1- Produtos Noutras Linguagens

Em C:

```
struct {  
    A first;  
    B second;  
};
```

Em Pascal:

```
record  
    first: A;  
    second: B  
end;
```

2- Coprodutos Noutras Linguagens

Em C:

```
struct {
    int tag; /* 1,2 */
    union {
        A ifA;
        B ifB;
    } data;
};
```

Em Pascal:

```
record
    case
        tag: integer
        of x =
            1: (P:A);
            2: (S:B)
    end;
```

3- O Isomorfismo swap

$$\langle \pi_2, \pi_1 \rangle$$

$$\begin{array}{ccccc} & & B \times A & & \\ & \swarrow \pi_2 & \downarrow & \searrow \pi_1 & \\ B & & \langle \pi_2, \pi_1 \rangle & & A \\ & \nearrow \pi_1 & & \nwarrow \pi_2 & \\ & & A \times B & & \end{array}$$

Exercício: provar que swap é a sua própria inversa.
É pois um isomorfismo $A \times B \cong B \times A$

Exercício: estudar o isomorfismo coswap = $[i_2, i_1]$

4- O Isomorfismo assocr

$$A \times (B \times C) \xleftarrow{\text{assocr}} (A \times B) \times C$$

$$\text{assocr} \stackrel{\text{def}}{=} \langle \pi_1 \bullet \pi_1, \langle \pi_2 \bullet \pi_1, \pi_2 \rangle \rangle$$

Exercício: converter para PW

Exercício: provar $\text{assocr} \bullet \text{assocl} = id$
para isso conjecturar assocl...

5- Lei da Troca

Sejam

$$B \xleftarrow{f} A, D \xleftarrow{g} A, B \xleftarrow{h} C \text{ and } D \xleftarrow{k} C$$

Então existe uma combinação de f,g,h,k que pode ser escrita como *split* ou como *either*:

$$[\langle f, g \rangle, \langle h, k \rangle] = \langle [f, h], [g, k] \rangle$$

6- O Isomorfismo undistr

$$A \times (B + C) \xleftarrow{\text{undistr}} (A \times B) + (A \times C)$$

$$\text{undistr} \stackrel{\text{def}}{=} [id \times i_1, id \times i_2]$$

Exercício: exprimir como *split* directamente,
e recorrendo à lei da troca

VI - Propriedades Universais e Condicionais - Guardas

1- Propriedades Universais

Apresentou-se um conjunto de leis de cálculo e combinadores de funções, com base em constructores de tipos como o produto e o coproduto

- Porquê estes?
- Qual a justificação teórica para as respectivas leis?

A resposta está na existência de uma *propriedade universal* que garante a **unicidade** dos combinadores, e a partir da qual se pode derivar as restantes leis.

Exemplo: produto

$$\begin{array}{ccccc} & A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} B \\ & \swarrow f & & \uparrow \langle f, g \rangle & \searrow g \\ C & & & & \end{array}$$

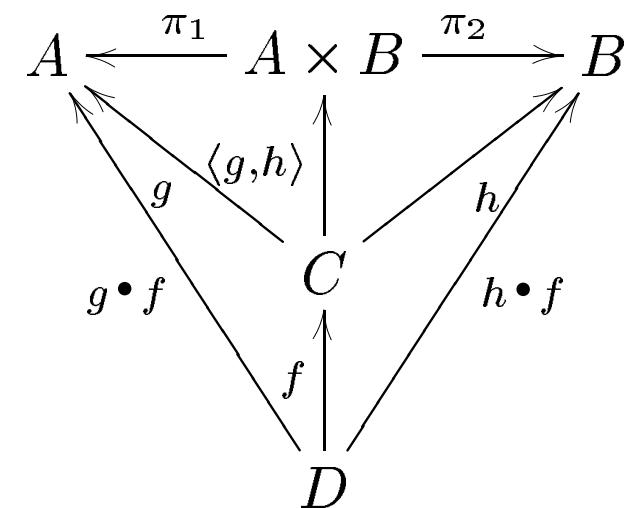
$$k = \langle f, g \rangle \Leftrightarrow \begin{cases} \pi_1 \cdot k = f \\ \pi_2 \cdot k = g \end{cases}$$

$\langle f, g \rangle$ existe, faz comutar o diagrama, e é única

Cancelamento é uma consequência imediata.

Como se pode provar a lei da fusão do produto?

$$\langle g, h \rangle \bullet f = \langle g \bullet f, h \bullet f \rangle$$



Uma propriedade que será muito útil futuramente:

Igualdade de *eithers*

$$[i, j] = [f, g] \Leftrightarrow \begin{cases} i = f \\ j = g \end{cases}$$

Exercício: provar, depois de escrever a propriedade universal dos coprodutos...

2- Expressões Condicionais: Guardas

Uma construção essencial numa linguagem de programação é a expressão *condicional*:

$$\text{if } (p\ x) \text{ then } (g\ x) \text{ else } (h\ x)$$
$$\begin{array}{c} \text{Bool} \xleftarrow{p} A \\ B \xleftarrow{g} A \\ B \xleftarrow{h} A \end{array}$$

significando $\left\{ \begin{array}{l} px \Rightarrow g\ x \\ \neg(px) \Rightarrow h\ x \end{array} \right.$

Como expressar esta noção no cálculo?
Introduz-se a noção de *guarda*:

Dado um predicado $\text{Bool} \xleftarrow{p} A$
define-se a guarda $A + A \xleftarrow{p?} A$
a ele associada como:

$$(p?)a = \begin{cases} pa & \Rightarrow i_1 a \\ \neg(pa) & \Rightarrow i_2 a \end{cases}$$

A guarda associada a um predicado é
mais informativa do que ele. Porquê?

Uma expressão condicional pode então ser escrita simplesmente como

$$[g, h] \cdot p?$$

Por exemplo a função que devolve um número caso ele seja par, ou o seu dobro caso seja ímpar:

```
f = [id, (2*)] . par?
```

```
par? x | (mod x 2) == 0 = i1 x  
        | otherwise       = i2 x
```

3- O Condicional de McCarthy

Sintaxe alternativa para as guardas:

$$p \rightarrow g, h \quad \stackrel{\text{def}}{=} \quad [g, h] \bullet p?$$

Exercício: provar a propriedade de fusão do condicional:

$$f \bullet (p \rightarrow g, h) \quad = \quad p \rightarrow f \bullet g, f \bullet h$$

VII - Tipos-exponencial e curry de funções

1- Tipos-exponencial

- Exponenciação de dois tipos:

$$B^A \stackrel{\text{def}}{=} \{g \mid g : B \longleftrightarrow A\}$$

- Contém informação sobre formas de mapear elementos de A em elementos de B

- Informação de mapeamento pode ser recuperada através da função de aplicação:
(análoga a (\$) em Haskell)

$$ap : B \xleftarrow{ap} B^A \times A$$
$$ap(f, a) \stackrel{\text{def}}{=} f a$$

2 - O Combinador Curry

$$\bar{f} : C \rightarrow A \rightarrow B$$

$$(\bar{f} c) a = f (c, a)$$

$$\text{ou } \bar{f} c = \lambda a \rightarrow f (c, a) \quad f : C \times A \rightarrow B$$

Ideia: transformar uma função de dois argumentos numa versão transposta (*curried*), de um só argumento, que devolve outra função

3- Prop: Cancelamento

$$f = ap \bullet (\bar{f} \times id)$$

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

comuta

Fácil de justificar com base nas definições *pointwise*

4- Propr: Fusão

$$\begin{array}{ccc} B^A & & B^A \times A \xrightarrow{ap} B \\ \uparrow \bar{g} & & \uparrow \bar{g} \times id \\ C & & C \times A \\ \uparrow f & & \uparrow f \times id \\ D & & D \times A \end{array}$$
$$g \bullet (f \times id) \quad \overline{g \bullet (f \times id)} = \bar{g} \bullet f$$

Fácil de justificar com base nas definições *pointwise*

5- Prop: Reflexão

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

$$\begin{array}{ccc} B^A & & B^A \times A \xrightarrow{ap} B \\ id_{B^A} \uparrow & id_{B^A} \times id_A \uparrow & \nearrow ap \\ B^A & & B^A \times A \end{array}$$

Fácil de justificar com base nas definições *pointwise*

6- O Combinador Exponenciação

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

$$f^A \stackrel{\text{def}}{=} \overline{f \cdot ap}$$

$$C \xleftarrow{f} B$$

Ideia: função que recebe uma outra e a compõe com f , como se entende da def. pointwise:

$$f^A g = f \cdot g$$

7- Propr: Absorção

$$\begin{array}{ccc} D^A & D^A \times A \xrightarrow{ap} & D \\ f^A \uparrow & f^A \times id \uparrow & f \uparrow \\ B^A & B^A \times A \xrightarrow{ap} & B \\ \bar{g} \uparrow & \bar{g} \times id \uparrow & g \nearrow \\ C & C \times A & \end{array}$$

$$\overline{f \bullet g} = f^A \bullet \bar{g}$$

Exercício: demonstrar esta propriedade

8- Proprs. Functoriais

$$(g \bullet h)^A = g^A \bullet h^A$$

$$id^A = id$$

9- Isomorfismos

O operador *curry* tem inversa, e testemunha o seguinte:

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

Outros isomorfismos:

$$A^{B+C} \cong A^B \times A^C$$

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

Quais as funções que testemunham estes isos?

VIII - Tipos Finitários e Elementares

1- Produtos Finitários

Produto n -ário: (n projecções π_i)

$$\prod_{i=1}^n A_i = A_1 \times \dots \times A_n$$

Split n -ário:

$$\langle f_1, \dots, f_n \rangle : A_1 \times \dots \times A_n \longrightarrow B$$

$$\text{com } f_i : A_i \longrightarrow B, i = 1, n$$

2- Coprodutos Finitários

Coproduto n -ário: (n injecções i_j)

$$\sum_{j=1}^n A_j = A_1 + \dots + A_n$$

Either n -ário:

$$[f_1, \dots, f_n] : A_1 + \dots + A_n \longrightarrow B$$

com $f_i : B \longleftarrow A_i$, $i = 1, n$.

3- Tipos de Dados Elementares

Tipos de Dados “enumerados”:

- Tipo 0 - tipo vazio, sem qualquer construtor
- Tipo 1 - tipo com um único elemento (construtor constante). É de facto uma classe de tipos isomorfos, uma vez que a escolha do construtor é irrelevante.

Em Haskell: `() : ()`

Alguns Isomorfismos envolvendo
Tipos de Dados elementares:

$$A + 0 \cong A$$

$$A \times 0 \cong 0$$

$$A \times 1 \cong A$$
 qual o iso?

$$A^0 \cong 1 \text{ (função vazia)}$$

$$\begin{array}{c} 1^A \\ A^1 \end{array} \cong \begin{array}{c} 1 \\ A \end{array} \text{ (funções constante)}$$

4- *Splits* e *Eithers* de aridade 1

para $n=1$ tem-se:

$$\langle f \rangle = [f] = f$$

$$\pi_1 = i_1 = id.$$

5- *Splits* e *Eithers* de aridade 0

para $n=0$ tem-se que a definição de Produto e Coproduto *degenera* nos tipos 1 e 0 resp:

$$\begin{array}{ccc} 1 & & 0 \\ \uparrow & & \downarrow \\ \Diamond & & [] \\ C & & C \end{array}$$

Justificado pela unicidade de *splits* e *eithers* !

6- O Tipo 1+A

Seja $B \xleftarrow{f} 1 + A$

Esta função será da forma $[b_0, g]$

com $b_0 \in B$ e $B \xleftarrow{g} A$

Análogo ao tratamento de “apontadores” numa linguagem imperativa!

Corresponde à estrutura do tipo **Maybe** a em Haskell.

7- O Tipo 2

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

Esta função será da forma $[\underline{b_1}, \underline{b_2}]$

Em particular se $B = \{b_1, b_2\}$ (for $b_1 \neq b_2$) f poderá ser um isomorfismo. De facto, todos os conjuntos de dois elementos são isomorfos:

$$1 + 1 \cong 2$$

Um tipo desta classe é $\text{Bool} = \{\text{TRUE}, \text{FALSE}\}$

8- O Tipo n

Pensamos no tipo n como representando a classe de todos os tipos contendo n elementos:

$$n \cong \underbrace{1 + \cdots + 1}_n$$

Isomorfismos relevantes:

$$\underbrace{A \times \cdots \times A}_n \cong A^n$$

$$\underbrace{A + \cdots + A}_n \cong n \times A$$

Exercício:

Quais as funções que testemunham os seguintes isomorfismos?

$$2 \times A \cong A + A$$

$$A \times A \cong A^2$$

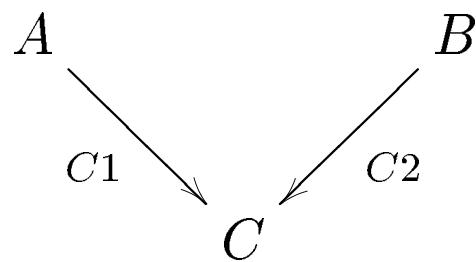
IX - Tipos de Dados Indutivos; Álgebras e Coálgebras

1- Álgebras e Co-álgebras

Considere-se a seguinte declaração de um tipo de dados numa linguagem funcional típica:

```
data C = C1 A | C2 B
```

Então os construtores c_1 e c_2 são as únicas formas disponíveis para construir valores do tipo C :



- Pode-se pois construir o diagrama:

$$\begin{array}{ccccc} & & A+B & & \\ A & \xrightarrow{i_1} & & \xleftarrow{i_2} & B \\ & \searrow c_1 & \downarrow [c_1, c_2] & \swarrow c_2 & \\ & & C & & \end{array}$$

- Então uma declaração de tipo corresponde a um *either* de constructores
- Por outro lado, a declaração garante a existência de um *isomorfismo* entre C e $A+B$, uma vez que não há outras formas de obter elementos de C

Cálculo da inversa inv de $[C1, C2]$

$$inv \bullet [C1, C2] = id$$

\leftrightarrow { by *+fusion* (1.40)}

$$[inv \bullet C1, inv \bullet C2] = id$$

\leftrightarrow { by *+reflexion* (1.39)}

$$[inv \bullet C1, inv \bullet C2] = [i_1, i_2]$$

\leftrightarrow { *either* uniqueness (1.58) }

$$inv \bullet C1 = i_1 \wedge inv \bullet C2 = i_2$$

Com variáveis, $inv : C \longrightarrow A + B$

$$inv(C1 a) = i_1 a$$

$$inv(C2 a) = i_2 b$$

A função $[C1, C2]$ permite construir valores do tipo C e designa-se *Álgebra* de C . A função inv é a sua *Coálgebra*. O par estabelece um isomorfismo:

$$\begin{array}{ccc} C & \begin{matrix} \xrightarrow{\hspace{2cm} inv \hspace{2cm}} \\ \cong \\ [C1, C2] \end{matrix} & A + B \end{array}$$

Caso particular, análogo a um apontador:

```
data C = C1 () | C2 B
```

Também pode ser definido (em Haskell) como:

```
data C = C1 | C2 B
```

Qual a álgebra / co-álgebra de *C*?

2- Álgebras de Tipos Recursivos

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

A estrutura indutiva deste tipo de dados é captada pelo seguinte isomorfismo:

$$LTree\ a \cong a + (LTree\ a) \times (LTree\ a)$$

O *either* dos constructores constitui de novo uma álgebra *in* para o tipo de dados.

$$in = [Leaf, Node]$$

$$in = [Leaf, Node]$$
$$out(Leaf x) = i_1 x$$
$$out(Node(l,r)) = i_2(l,r)$$

Torna-se assim possível definir funções sobre tipos de dados indutivos no cálculo:

$$height = [\underline{0}, succ \cdot max \cdot (height \times height)] \cdot outLTree$$

(Função que calcula a altura de uma árvore)

3- Caracterização Equacional de Tipos Recursivos

Lista ligada em C

```
typedef struct N {  
    A first;  
    struct N *next;  
} *L;
```

$$\begin{cases} L = 1 + N \\ N = A \times (1 + N) \end{cases} \quad \text{OU} \quad \begin{cases} L = 1 + N \\ N = A \times L \end{cases}$$

Eliminando N obtém-se $L = 1 + A \times L$

Listas em Haskell

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

(A um qualquer tipo constante)

$$L = 1 + A \times L$$

4- Resolução de Equações Recursivas

$$L = 1 + A \times L$$

\leftrightarrow { substitution of $1 + A \times L$ for L }

$$L = 1 + A \times (1 + A \times L)$$

\leftrightarrow { distributive property (1.50) }

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

\leftrightarrow { unit of product (1.79) and associativity of product (1.32) }

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

$$\begin{aligned}
L &\cong 1 + A + (A \times A) \times L \\
\Leftrightarrow & \quad \{ \text{ by (1.80), (1.82) and (1.85)} \} \\
L &\cong A^0 + A^1 + A^2 \times L \\
\Leftrightarrow & \quad \{ \text{ another substitution as above and similar simplifications} \} \\
L &\cong A^0 + A^1 + A^2 + A^3 \times L \\
\Leftrightarrow & \quad \{ \text{ after } (n+1)\text{-many similar steps} \} \\
L &\cong \sum_{i=0}^n A^i + A^{n+1} \times L
\end{aligned}$$

mas então, quando $n \rightarrow \infty$ tem-se

$$L \quad \approx \quad \sum_{i=0}^{\infty} A^i$$

e

$$L \quad \approx \quad A^*$$

O tipo L é isomorfo ao tipo que contém todas as sequências finitas de elementos do tipo A

(como seria de esperar !?)