

# Lambda-calculus: fundamentos de Programação Funcional

Mário Florido

24 de novembro de 2021

# Lambda-calculus

- ▶ Modelo computação universal (i.e., equipotente à máquina de Turing)
- ▶ Ao contrário da MT, o cálculo- $\lambda$  é também um modelo para linguagens de programação:
  - ▶ âmbito de variáveis
  - ▶ ordem de computação
  - ▶ estruturas de dados
  - ▶ recursão
  - ▶ ...
- ▶ As linguagens funcionais são implementações computacionais do cálculo- $\lambda$  (Landin, 1964).

# Bibliografia

1. *Foundations of Functional Programming*, notas de um curso de Lawrence C. Paulson, Computer Laboratory, University of Cambridge.
2. *Lambda Calculi: a guide for computer scientists*, Chris Hankin, Graduate Texts in Computer Science, Oxford University Press.

# Plano

Sintaxe

Reduções

Computação

# Plano

Sintaxe

Reduções

Computação

# Termos do cálculo- $\lambda$

- $x, y, z, \dots$  uma variável é um termo;  
 $(\lambda x M)$  é um termo se  $x$  é variável e  $M$  é um termo;  
 $(M N)$  é um termo se  $M$  e  $N$  são termos.

## exemplos de termos

$y$   
 $(\lambda x y)$   
 $((\lambda x y) (\lambda x (\lambda x y)))$   
 $(\lambda y (\lambda x (y (y x))))$

## não são termos

$()$   
 $x \lambda y$   
 $x(y)$   
 $(\lambda x (\lambda y y))$

# Interpretação do cálculo- $\lambda$

$(\lambda x M)$  função com parâmetro  $x$  e corpo  $M$ .

$(M N)$  aplicação de  $M$  ao argumento  $N$ .

Exemplos:

$(\lambda x x)$  *função identidade*: a  $x$  faz corresponder  $x$

$(\lambda x (\lambda y x))$

- ▶ Não há distinção entre dados e programas;
- ▶ Não há constantes (e.g. números).
- ▶ Tudo são  $\lambda$ -termos!

# Convenção de parêntesis

Abreviaturas:

$$\lambda x_1 x_2 \dots x_n. M \quad \equiv \quad (\lambda x_1 (\lambda x_2 \dots (\lambda x_n M) \dots))$$

$$(M_1 M_2 \dots M_n) \quad \equiv \quad (\dots (M_1 M_2) \dots M_n)$$

Exemplos:

$$\lambda xy. x \quad \equiv \quad (\lambda x (\lambda y x))$$

$$\lambda xyz. xz(yz) \quad \equiv \quad (\lambda x (\lambda y (\lambda z ((x z) (y z)))))$$



# Substituição

$M[N/x]$  o termo resultante da substituição de  $x$  em  $M$  por  $N$ .

$$(\lambda x y)[(z z)/y] \equiv (\lambda x (z z))$$

$$(\lambda x y)[(z z)/x] \equiv (\lambda x y)$$

# Plano

Sintaxe

Reduções

Computação

# Conversão- $\beta$

$$((\lambda x M) N) \rightarrow_{\beta} M[N/x] \quad \text{se } BV(M) \cap FV(N) = \emptyset$$

Exemplo:

$$((\lambda x \underbrace{(x x)}_M) \underbrace{(y z)}_N) \rightarrow_{\beta} (x x)[(y z)/x] \equiv ((y z) (y z))$$

# Conversão- $\beta$

$$((\lambda x M) N) \rightarrow_{\beta} M[N/x] \quad \text{se } BV(M) \cap FV(N) = \emptyset$$

Exemplo:

$$((\lambda x \underbrace{(x x)}_M) \underbrace{(y z)}_N) \rightarrow_{\beta} (x x)[(y z)/x] \equiv ((y z) (y z))$$

Corresponde à invocação de uma função:

- ▶  $x$  o parâmetro;
- ▶  $M$  o corpo da função;
- ▶  $N$  o argumento.

# Currying

Não necessitamos de abstracções de duas ou mais variáveis:

$$\lambda xy. M \equiv (\lambda x (\lambda y M))$$

Substituímos os argumentos um de cada vez:

$$\begin{aligned} ((\lambda xy. M) P Q) &\equiv (((\lambda x (\lambda y M)) P) Q) \\ &\rightarrow_{\beta} ((\lambda y M)[P/x] Q) \\ &\rightarrow_{\beta} M[P/x][Q/y] \end{aligned}$$

Esta codificação chama-se “*currying*” em homenagem a Haskell Curry.

# Reduções

Escrevemos  $M \rightarrow N$  se  $M$  **reduz num passo** a  $N$  usando conversão  $\beta$ .

Escrevemos  $M \rightarrow^* N$  para a **redução em múltiplos passos** ( $\rightarrow^*$ ).

# Forma normal

Se não existir  $N$  tal que  $M \rightarrow N$ , então  $M$  está em **forma normal**.

$M$  **admite forma normal**  $N$  se  $M \rightarrow^* N$  e  $N$  está em forma normal.

Exemplo:

$$(\lambda x. a x) ((\lambda y. by) c) \rightarrow a((\lambda y. by) c) \rightarrow a(bc) \not\rightarrow$$

Logo:  $(\lambda x. a x) ((\lambda y. by) c)$  admite forma normal  $a(bc)$ .

Analogia: resultado de uma computação.

# Termos sem forma normal

Nem todos os termos admitem forma normal:

$$\begin{aligned}\Omega &\equiv ((\lambda x. x x) (\lambda x. x x)) \\ &\rightarrow_{\beta} (x x)[(\lambda x. x x)/x] \\ &\equiv ((\lambda x. x x) (\lambda x. x x)) \equiv \Omega\end{aligned}$$

Logo:

$$\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \dots$$

Analogia: uma computação que não termina.



# Confluência

Podemos efectuar reduções por ordens diferentes.

Exemplo:

$$\underline{(\lambda x. a x) ((\lambda y. b y) c)} \rightarrow a(\underline{(\lambda y. b y) c}) \rightarrow a(bc) \not\rightarrow$$

$$(\lambda x. a x) (\underline{(\lambda y. b y) c}) \rightarrow \underline{(\lambda x. a x) (bc)} \rightarrow a(bc) \not\rightarrow$$

**P:** Será que chegamos sempre à mesma forma normal?

# Confluência

Podemos efectuar reduções por ordens diferentes.

Exemplo:

$$\underline{(\lambda x. a x) ((\lambda y. b y) c)} \rightarrow a(\underline{(\lambda y. b y) c}) \rightarrow a(bc) \not\rightarrow$$

$$(\lambda x. a x) (\underline{(\lambda y. b y) c}) \rightarrow \underline{(\lambda x. a x) (bc)} \rightarrow a(bc) \not\rightarrow$$

P: Será que chegamos sempre à mesma forma normal?

R: Sim (**Teorema de Church-Rosser**)

# Estratégias de redução

Como reduzir  $(M N) \rightarrow P$ ?

**ordem normal:** reduzir  $M$  e substituir  $N$  sem reduzir.

1.  $M \rightarrow (\lambda x M')$
2.  $M'[N/x] \rightarrow P$

**ordem aplicativa:** reduzir  $M$  e  $N$  antes de fazer a substituição da variável.

1.  $M \rightarrow (\lambda x M')$
2.  $N \rightarrow N'$
3.  $M'[N'/x] \rightarrow P$

# Alguns factos sobre reduções

1. Se a redução por ambas as estratégias termina, então chegam à mesma forma normal
2. Se um termo admite forma normal, esta pode sempre obtida pela redução em ordem normal
3. A redução em ordem applicativa pode não terminar mesmo quando existe forma normal
4. A redução em ordem normal pode reduzir o mesmo termo várias vezes

# Ordem aplicativa: não terminação

Seja  $\Omega \equiv ((\lambda x. x x) (\lambda x. x x))$ ; vamos reduzir

$$(\lambda x. y) \Omega$$

Redução em ordem normal

$$\underline{((\lambda x. y) \Omega)} \rightarrow_{\beta} y \quad \text{forma normal}$$

Redução em ordem aplicativa

$$((\lambda x. y) \underline{\Omega}) \rightarrow_{\beta} ((\lambda x. y) \underline{\Omega}) \rightarrow_{\beta} \cdots \quad \text{não termina}$$

# Ordem normal: computação redundante

Supondo **mult** um termo tal que

$$\mathbf{mult} \ N \ M \rightarrow N \times M$$

para codificações  $N$ ,  $M$  de números naturais (veremos como mais à frente).

Definindo

$$\mathbf{sqr} \equiv \lambda x. \mathbf{mult} \ x \ x$$

vamos reduzir

$$\mathbf{sqr} \ (\mathbf{sqr} \ N)$$

# Ordem normal: computação redundante

Redução em ordem aplicativa:

$$\mathbf{sqr}(\mathbf{sqr} N) \rightarrow \mathbf{sqr}(\mathbf{mult} N N) \rightarrow \mathbf{sqr} N^2 \rightarrow \mathbf{mul} N^2 N^2$$

Redução em ordem normal:

$$\begin{aligned} \mathbf{sqr}(\mathbf{sqr} N) &\rightarrow \mathbf{mult}(\mathbf{sqr} N)(\mathbf{sqr} N) \\ &\rightarrow \mathbf{mult}(\underbrace{\mathbf{mult} N N \mathbf{mult} N N}_{\text{duplicação}}) \end{aligned}$$

# Plano

Sintaxe

Reduções

Computação



# Computação usando cálculo- $\lambda$

O cálculo- $\lambda$  é um modelo de **computação universal**: qualquer função recursiva (computável por uma máquina de Turing) pode ser codificada no cálculo- $\lambda$ .

# Computação usando cálculo- $\lambda$

Estruturas de dados como booleanos, inteiros, listas, etc. não são primitivas do cálculo- $\lambda$ .

Estas estruturas podem ser definidas usando apenas o cálculo puro.

Contudo: implementações de linguagens funcionais usam representações otimizadas por razões de eficiência.

# Valores booleanos

Definimos:

$$\mathbf{true} \equiv \lambda xy. x$$
$$\mathbf{false} \equiv \lambda xy. y$$
$$\mathbf{if} \equiv \lambda pxy. pxy$$

Então:

$$\mathbf{if\ true\ } M\ N \rightarrow\!\!\rightarrow M$$
$$\mathbf{if\ false\ } M\ N \rightarrow\!\!\rightarrow N$$

Exercício: verificar as reduções acima.

# Pares ordenados

Um *constructor* e dois *selectores*:

**pair**  $\equiv \lambda xyf. fxy$

**fst**  $\equiv \lambda p. p \text{ true}$

**snd**  $\equiv \lambda p. p \text{ false}$

Temos:

$$\begin{aligned} \mathbf{fst} (\mathbf{pair} \ M \ N) &\rightarrow \mathbf{fst} (\lambda f. f \ M \ N) \\ &\rightarrow (\lambda f. f \ M \ N) \ \mathbf{true} \\ &\rightarrow \mathbf{true} \ M \ N \\ &\rightarrow M \end{aligned}$$

Analogamente: **snd** (**pair** *M* *N*)  $\rightarrow N$ .

# Codificar números naturais

Usando numerais de Church:

$$\underline{0} \equiv \lambda f x. x$$

$$\underline{1} \equiv \lambda f x. f x$$

$$\underline{2} \equiv \lambda f x. f (f x)$$

$\vdots$

$$\underline{n} \equiv \lambda f x. \underbrace{f (\dots (f x) \dots)}_{n \text{ vezes}}$$

Intuição:  $\underline{n}$  itera uma função  $n$  vezes.

# Aritmética

$$\mathbf{succ} \equiv \lambda nfx. f (n f x)$$

$$\mathbf{iszero} \equiv \lambda n. n (\lambda x. \mathbf{false}) \mathbf{true}$$

$$\mathbf{add} \equiv \lambda mnfx. m f (n f x)$$

Verificar:

$$\mathbf{succ} \, \underline{n} \rightarrow \underline{n + 1}$$

$$\mathbf{iszero} \, \underline{0} \rightarrow \mathbf{true}$$

$$\mathbf{iszero} \, (\underline{n + 1}) \rightarrow \mathbf{false}$$

$$\mathbf{add} \, \underline{n} \, \underline{m} \rightarrow \underline{n + m}$$

Analogamente: subtração, multiplicação, expoente, etc.

# Listas

$$[x_1, x_2, \dots, x_n] \simeq \mathbf{cons} \ x_1 \ (\mathbf{cons} \ x_2 \ (\dots (\mathbf{cons} \ x_n \ \mathbf{nil}) \dots))$$

Dois constructores, teste da lista vazia e dois selectores:

$$\mathbf{nil} \equiv \lambda z. z$$

$$\mathbf{cons} \equiv \lambda xy. \mathbf{pair} \ \mathbf{false} \ (\mathbf{pair} \ x \ y)$$

$$\mathbf{null} \equiv \mathbf{fst}$$

$$\mathbf{hd} \equiv \lambda z. \mathbf{fst} \ (\mathbf{snd} \ z)$$

$$\mathbf{tl} \equiv \lambda z. \mathbf{snd} \ (\mathbf{snd} \ z)$$

# Listas

Verificar:

$$\text{null nil} \rightarrow \text{true} \quad (1)$$

$$\text{null (cons } M N) \rightarrow \text{false} \quad (2)$$

$$\text{hd (cons } M N) \rightarrow M \quad (3)$$

$$\text{tl (cons } M N) \rightarrow N \quad (4)$$

NB: (2), (3), (4) resultam das propriedades de pares, mas (1) não.



# Variáveis locais

**let**  $x = M$  **in**  $N$

Exemplo:

**let**  $f = \lambda x. \mathbf{add} \ x \ x$   
**in**  $\lambda x. f \ (f \ x)$

# Tradução para o cálculo- $\lambda$

Definimos:

$$\mathbf{let } x = M \mathbf{ in } N \equiv (\lambda x. N) M$$

Então:

$$\mathbf{let } x = M \mathbf{ in } N \rightarrow N[M/x]$$

# Declarações imbricadas

Não necessitamos de sintaxe extra:

$$\begin{aligned} & \mathbf{let} \{x = M; y = N\} \mathbf{in} P \\ \equiv & \mathbf{let} x = M \mathbf{in} (\mathbf{let} y = N \mathbf{in} P) \end{aligned}$$

# Declarações recursivas

Tentativa:

```
let  $f = \lambda x.$  if (iszero  $x$ ) 1 (mult  $x$  (f (sub  $x$  1)))  
in  $f$  5
```

# Declarações recursivas

Tentativa:

```
let  $f = \lambda x.$  if (iszero  $x$ ) 1 (mult  $x$  ( $f$  (sub  $x$  1)))  
in  $f$  5
```

Tradução:

```
 $(\lambda f. f \text{ 5}) (\lambda x. \text{if } (\text{iszero } x) \text{ 1 } (\text{mult } x (\textcolor{red}{f} (\text{sub } x \text{ 1}))))$ 
```

Não define uma função recursiva porque  $\textcolor{red}{f}$  ocorre livre no corpo da definição.

# Declarações recursivas: combinadores ponto-fixado

Solução: usar um **combinador ponto-fixado** i.e. um termo **Y** tal que

$$Y F = F (Y F) \quad \text{para qualquer termo } F$$

Definimos o factorial recursivo como:

```
let f = Y (λg x. if (iszero x) 1 (mult x (g (sub x 1))))  
in f 5
```

Note que *g* ocorre ligada no corpo da função.

# Definições recursivas: combinador ponto-fixado

Seja:

$Y F = F (Y F)$  para qualquer  $M$

$\text{fact} \equiv Y (\lambda g x. \text{if } (\text{iszero } x) \ 1 \ (\text{mult } x \ (g \ (\text{sub } x \ 1))))$

Calculemos:

$$\begin{aligned} \text{fact } \underline{5} &\equiv Y (\lambda g x. \dots) \underline{5} \\ &= (\lambda g x. \dots) \underbrace{(Y (\lambda g x. \dots))}_{\text{fact}} \underline{5} \\ &\rightarrow \text{if } (\text{iszero } \underline{5}) \ 1 \ (\text{mult } \underline{5} \ (\text{fact } (\text{sub } \underline{5} \ 1))) \\ &\rightarrow \text{if } \text{false } \underline{1} \ (\text{mult } \underline{5} \ (\text{fact } \underline{4})) \\ &\rightarrow \text{mult } \underline{5} \ (\text{fact } \underline{4}) \\ &\rightarrow \text{mult } \underline{5} \ (\text{mult } \underline{4} \ (\dots (\text{mult } \underline{1} \ \underline{1}) \dots)) \equiv \underline{120} \end{aligned}$$

# Combinadores ponto-fixo

**Y** pode ser definido no cálculo- $\lambda$  puro (Haskell B. Curry):

$$\mathbf{Y} \equiv \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

Verificação:

$$\begin{aligned}\mathbf{Y} F &\rightarrow (\lambda x. F (xx)) (\lambda x. F (xx)) \\ &\rightarrow F ((\lambda x. F (xx)) (\lambda x. F (xx))) \\ &\leftarrow F (\mathbf{Y} F)\end{aligned}$$

Logo

$$\mathbf{Y} F = F (\mathbf{Y} F)$$

Há uma infinidade de outros combinadores ponto-fixo (ver a bibliografia).