

Lambda Calculus

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2021/2022

Lambda calculus

- Formalismo introduzido na década de 1930 por Alonzo Church como uma maneira de formalizar o conceito de computabilidade efectiva.
- O λ -calculus forneceu uma base teórica sólida para as linguagens de programação funcionais.
- O λ -calculus é a menor linguagem de programação que existe.
 - ▶ O λ -calculus é universal no sentido de que qualquer função computável pode ser expressa e avaliada usando esse formalismo. É, portanto, equivalente às máquinas de Turing.
 - ▶ No entanto, o λ -calculus enfatiza o uso de regras de transformação e não se importa com a máquina real que as implementa.

Lambda calculus

- O λ -calculus **puro** lida apenas com variáveis, definição de funções, e aplicação de funções.
- Se acrescentarmos outros tipos de dados e operações (tal com inteiros e adição) temos um λ -calculus **aplicado**.
- No λ -calculus puro não há distinção de tipos. Todas as expressões pertencem a **um único universo**.
- Do ponto de vista computacional o λ -calculus puro é mais simples de descrever, mas do ponto de vista matemático e lógico é o contrário:
 - ▶ o λ -calculus **tipificado** pode ser entendido em termos das funções matemáticas usuais,
 - ▶ enquanto o puro levanta problemas em termos fundacionais.

Sintaxe

Termos

- Assume-se um conjunto enumerável de variáveis: x, y, z, \dots
- As expressões (ou termos) do lambda calculus puro têm a seguinte sintaxe abstracta

$$e ::= x \mid \lambda x. e \mid e_1 e_2$$

denominadas **variáveis**, **abstrações** e **aplicações**, respectivamente.

Uma expressão pode estar entre parêntesis mas, para os evitar, segue-se a seguinte convenção:

- a aplicação é associativa à esquerda;
- o âmbito da abstração λ estende-se para a direita o mais possível.

- $abcd$ em vez de $((ab)c)d$
- $\lambda x. \lambda b. f x (\lambda z. b z)$ em vez de $\lambda x. (\lambda b. ((f x) (\lambda z. b z)))$
- $(\lambda y. \lambda x. x (y a) y) (\lambda z. f z)$ em vez de $(\lambda y. (\lambda x. (x (y a) y))) (\lambda z. f z)$

Variáveis livres e ligadas

$FV(e)$ denota o conjunto das *variáveis livres* de uma expressão e

$$\begin{aligned}FV(x) &= \{x\} \\FV(\lambda x. a) &= FV(a) \setminus \{x\} \\FV(a b) &= FV(a) \cup FV(b)\end{aligned}$$

$BV(e)$ denota o conjunto das *variáveis ligadas* de uma expressão e

$$\begin{aligned}BV(x) &= \{\} \\BV(\lambda x. a) &= BV(a) \cup \{x\} \\BV(a b) &= BV(a) \cup BV(b)\end{aligned}$$

Variáveis livres e ligadas

- Uma variável x diz-se *livre* em e se $x \in FV(e)$.
- Uma variável x diz-se *ligada* em e se $x \in BV(e)$.
- Uma expressão sem variáveis livres diz-se *fechada* (ou *combinador*).

Uma variável pode ser simultaneamente livre e ligada numa dada expressão. Por exemplo,

- $(x y) \lambda z. \lambda x. x z$
- $\lambda x. x (\lambda y. y a) x y$

Semântica

- O significado das expressões é dado operacionalmente através de uma regra de redução de capta o efeito de aplicar uma função ao seu argumento.

- Essa redução tem o nome de *redução β*

$$(\lambda x. a) b \rightarrow a[b/x]$$

onde $a[b/x]$ representa o termo obtido por *substituição das ocorrências livres de x em a por b* .

- A substituição é uma operação “melindrosa”.
 - ▶ Qual o efeito da substituição $(\lambda z. z x)[z/x]$?
 - ▶ Uma abordagem “naive” conduz ao problema de *captura de variáveis!*

Substituição (abordagem de Church)

Para evitar o problema de captura de variáveis livres, a definição original de Church para a substituição foi a seguinte:

$$\begin{aligned}x[a/x] &= a \\y[a/x] &= y && \text{se } x \neq y \\(\lambda x. b)[a/x] &= (\lambda x. b) \\(\lambda y. b)[a/x] &= (\lambda y. b[a/x]) && \text{se } x \notin FV(b) \text{ ou } y \notin FV(a) \\(\lambda y. b)[a/x] &= (\lambda z. (b[z/y])[a/x]) && \text{se } x \in FV(b) \text{ e } y \in FV(a) \\&&& \text{sendo } z \text{ uma variável fresca} \\(e_1 e_2)[a/x] &= (e_1[a/x]) (e_2[a/x])\end{aligned}$$

Conversão α

- Como sabemos, o nome das variáveis ligadas não é importante. Por exemplo,

$$\sum_{x=1}^{20} x \quad \text{e} \quad \sum_{y=1}^{20} y$$

descrevem o mesmo somatório.

- Church introduziu na apresentação original do λ -calculus uma noção de **conversão α** que traduz esta ideia.

α -conversão

$$\lambda x. e = \lambda y. e[y/x] \quad , \text{ se } y \notin \text{FV}(e)$$

Esta conversão induz uma relação de equivalência nos termos.

Convenção das variáveis

Para simplificar o tratamento formal do λ -calculus é usual trabalhar-se ao nível das classes de equivalência geradas pela α -conversão.

- Considera-se um λ -termo como um representante da sua classe de α -equivalência.
- Interpretamos $e[a/x]$ como uma operação na classe de equivalência, usando o representante que for conveniente de acordo com a seguinte convenção:

Convenção das variáveis

Todas as variáveis ligadas são escolhidas de forma a serem diferentes das variáveis livres.

Substituição

Assumindo a convenção das variáveis a definição de substituição fica simplificada:

Substituição

$$\begin{aligned} x[a/x] &= a \\ y[a/x] &= y \\ (\lambda y. b)[a/x] &= (\lambda y. b[a/x]) \\ (e_1 e_2)[a/x] &= (e_1[a/x]) (e_2[a/x]) \end{aligned} \quad \text{se } x \neq y$$

Lema da substituição

Sejam x e y variáveis distintas e $x \notin \text{FV}(e)$, então

$$(a[b/x])[e/y] = (a[e/y])[b[e/y]/x]$$

Prova: Por indução na estrutura de a .

β -redução

- A redução β indica o efeito de aplicar uma função a um argumento.

β -redução

A β -redução, \rightarrow_β , é definida como o fecho compatível da regra

$$(\lambda x. a) b \rightarrow_\beta a[b/x]$$

- \rightarrow_β^* é fecho reflexivo e transitivo de \rightarrow_β .
- $=_\beta$ é fecho reflexivo, simétrico e transitivo de \rightarrow_β .
- um termo $(\lambda x. a) b$ chama-se **β -redex** e a $a[b/x]$ o seu **contractum**

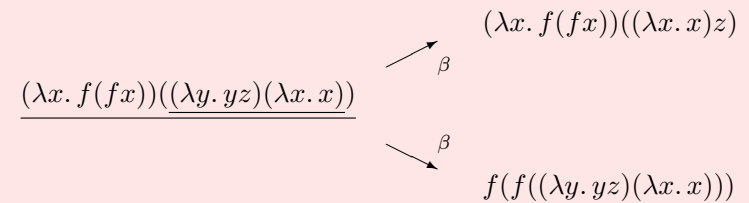
Uma expressão que não contém nenhum β -redex diz-se uma **forma normal**.

- Por **fecho compatível** entende-se que

$$\begin{aligned} \text{se } a \rightarrow_\beta a' & \quad \text{então } ab \rightarrow_\beta a'b \\ \text{se } b \rightarrow_\beta b' & \quad \text{então } ab \rightarrow_\beta ab' \\ \text{se } a \rightarrow_\beta a' & \quad \text{então } \lambda x. a \rightarrow_\beta \lambda x. a' \end{aligned}$$

β -redexes

Uma expressão pode ter mais do que um β -redex. Por exemplo:



Exercício

Apresente as possíveis sequências de redução dos termos

- $(\lambda x. (\lambda y. yx)z)(zw)$
- $(\lambda u. \lambda v. v)((\lambda x. xx)(\lambda x. xx))$

Confluência

Teorema (Church-Rosser)

Para qualquer expressão e , se $e \rightarrow_{\beta}^* e_1$ e $e \rightarrow_{\beta}^* e_2$, então existe uma expressão e' tal que $e_1 \rightarrow_{\beta}^* e'$ e $e_2 \rightarrow_{\beta}^* e'$.

Unicidade das normas normais.

A forma normal de uma expressão, se existir, é única.

Normalização

No que se refere à normalização, existem expressões e para as quais

- todas as sequências de redução com origem em e terminam

$$(\lambda x. f(fx))((\lambda x. x)z)$$

- nenhuma das sequências de redução com origem em e termina

$$(\lambda x. f(xx))(\lambda x. f(xx))$$

- apenas algumas das sequências de redução com origem em e terminam

$$(\lambda u. \lambda v. v)((\lambda x. xx)(\lambda x. xx))$$

Estratégias de redução

A normalização (se possível) de um termo e a quantidade de esforço necessária para isso, pode depender da estratégia de redução usada.

Chamamos *estratégia de redução* ao critério que é seguido para escolher o próximo β -redex a ser reduzido.

- **“Full beta-reduction”**: qualquer β -redex pode ser seleccionado.
- **“Normal-order reduction”**: o β -redex seleccionado é o mais à esquerda e mais externo.
- **“Applicative-order reduction”**: o β -redex seleccionado é o mais à esquerda e mais interno.

“Normal-order reduction”: *leftmost outermost* redex

Na **ordem normal de redução** escolhe-se sempre o redex mais à esquerda mais externo.

Exemplo de redução seguindo a ordem normal

$$\begin{aligned} & (\lambda x. x ((\lambda y. y) x)) ((\lambda a. a) (\lambda b. b)) \\ \rightarrow_{\beta} & ((\lambda a. a) (\lambda b. b)) ((\lambda y. y) ((\lambda a. a) (\lambda b. b))) \\ \rightarrow_{\beta} & (\lambda b. b) ((\lambda y. y) ((\lambda a. a) (\lambda b. b))) \\ \rightarrow_{\beta} & (\lambda y. y) ((\lambda a. a) (\lambda b. b)) \\ \rightarrow_{\beta} & (\lambda a. a) (\lambda b. b) \\ \rightarrow_{\beta} & (\lambda b. b) \end{aligned}$$

“Applicative-order reduction”: *leftmost innermost* redex

Na **ordem applicativa de redução** escolhe-se sempre o redex mais à esquerda mais interno.

Exemplo de redução seguindo a ordem applicativa

$$\begin{aligned} & (\lambda x. x ((\lambda y. y) x)) ((\lambda a. a) (\lambda b. b)) \\ \rightarrow_{\beta} & (\lambda x. x x) ((\lambda a. a) (\lambda b. b)) \\ \rightarrow_{\beta} & (\lambda x. x x) (\lambda b. b) \\ \rightarrow_{\beta} & (\lambda b. b) (\lambda b. b) \\ \rightarrow_{\beta} & (\lambda b. b) \end{aligned}$$

Normalização

Exercício

Considere os termos

$$\begin{aligned} I &= (\lambda x. x) \\ K &= (\lambda x. \lambda y. x) \\ S &= (\lambda x. \lambda y. \lambda z. x z (y z)) \\ \Omega &= ((\lambda x. x x) (\lambda x. x x)) \end{aligned}$$

Construa sequências de redução para as expressões abaixo, seguindo as diferentes estratégias de redução apresentadas.

- $S K K$
- $K S \Omega$
- $I (I (\lambda z. I z))$

Normalização

Uma expressão e diz-se *fortemente normalizável* se todas as sequências de redução com origem em e terminam; diz-se (*fracamente*) *normalizável* se existir alguma sequência de redução com origem em e que termina; e diz-se *não normalizável* se nenhuma sequência de redução com origem em e termina.

Será que existe alguma estratégia de redução que nos leve sempre a alcançar a forma normal de uma expressão, caso ela exista?

Sim, a “*normal-order reduction*”!

Teorema da standardização

Se existir alguma sequência de redução começada numa expressão e que termine, então sequência começada em e que segue a ordem normal de redução termina.

Formas canónicas

O processo de avaliação das linguagens de programação funcionais (LPFs) está intimamente ligado ao processo de redução de uma expressão à sua forma normal. Contudo, é um processo menos geral:

- Nas LPFs apenas se avaliam *expressões fechadas* (uma vez que um programa com variáveis livres não é bem formado).
- Em vez de reduzir uma expressão à sua forma normal, o processo de avaliação das LPFs termina assim que se obtém uma *forma canónica*.
 - ▶ A noção de forma canónica pode variar conforme a LPF em causa, mas inclui sempre as abstrações (que são as únicas formas canónicas do λ -calculus puro).

Formas canónicas

No λ -calculus puro uma *forma canónica* é uma abstracção $\lambda x. e$.

Usaremos a letra v como meta-variável para representar uma forma canónica (ou *valor*).

Apresente a sequência de redução que segue a ordem normal de redução para o termo

$$(\lambda x. x (\lambda y. x y y) x)(\lambda z. \lambda w. z)$$

Repare que a forma normal é atingida ao fim de 5 passos de redução, mas a primeira forma canónica é atingida ao fim de 3 passos.

Formas canónicas

Ordem normal de avaliação

Dada uma expressão fechada e , diz-se que e *avalia para* v , e escrevemos $e \Rightarrow v$, quando v é a primeira forma canónica da sequência de redução começada em e que segue a ordem normal de redução.

Se tal forma canónica v não existir, dizemos que e *diverge*, e denotamos isso por $e \uparrow$.

Ordem normal de avaliação

Para uma expressão fechada e existem 3 possibilidades:

- A sequência da ordem normal de redução de e termina numa forma normal. Portanto, a sequência de redução contém pelo menos uma forma canónica. Ex:

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

- A sequência da ordem normal de redução de e não termina, mas contém uma forma canónica. Ex:

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

- A sequência da ordem normal de redução de e não termina e não contém uma forma canónica. Ex:

$$(\lambda x. x x)(\lambda x. x x)$$

Semântica “call-by-name”

- A terminologia usual usada nas LPFs para a ordem normal de avaliação é “*call-by-name evaluation*” ou “*lazy evaluation*”.
- O processo de avaliação pode ser descrito operacionalmente no estilo “small-step” ou “big-step”.

Small-step semantics

$$(CBN1_{ss}) \quad (\lambda x. e_1) e_2 \rightarrow e_1[e_2/x]$$

$$(CBN2_{ss}) \quad \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

Semântica “call-by-name”

Big-step semantics

$$(CBN1_{bs}) \quad \lambda x. e \Rightarrow \lambda x. e$$

$$(CBN2_{bs}) \quad \frac{e_1 \Rightarrow \lambda x. e \quad e[e_2/x] \Rightarrow v}{e_1 e_2 \Rightarrow v}$$

- Na prática, a avaliação “call-by-name” proíbe reduções dentro de abstrações, e aplica as funções logo que possível (i.e., sem avaliar previamente o seu argumento).
- Variantes da avaliação “call-by-name” são usadas em linguagens de programação conhecidas, como por exemplo Haskell. O Haskell usa uma versão optimizada desta estratégia, chamada “*call-by-need*”, onde várias ocorrências do mesmo argumento são avaliadas apenas uma única vez (através de uma gestão de termos partilhados).

Semântica “call-by-value”

- Uma outra estratégia de avaliação muito usada nas LPFs consiste em efectivar a aplicação de uma função ao seu argumento apenas quando o argumento já foi reduzido a um valor (i.e., à sua forma canónica).
- Esta estratégia denomina-se de “*eager evaluation*” ou “*call-by-value evaluation*” (terminologia mais usual nas LPFs)
- O processo de avaliação pode ser descrito operacionalmente no estilo “small-step” ou “big-step”.

Semântica “call-by-value”

Small-step semantics

$$(CBV1_{ss}) \quad (\lambda x. e) v \rightarrow e[v/x]$$

$$(CBV2_{ss}) \quad \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

$$(CBV3_{ss}) \quad \frac{e \rightarrow e'}{v e \rightarrow v e'}$$

- Como podemos ver, dada uma aplicação $e_1 e_2$, primeiro avalia-se e_1 até obter um valor, depois avalia-se e_2 até obter um valor, e só no fim efectuamos a aplicação da função.

Semântica “call-by-value”

- Para apresentar a “eager evaluation” como uma semântica big-step, precisamos de introduzir a seguinte relação

Dado um termo fechado e , denotamos por $e \Rightarrow_E v$ o facto de existir uma sequência de redução de e para v , onde cada passo corresponde a uma redução do β_E -redex mais à esquerda que não é um subtermo de uma forma canónica.

Um β_E -redex é um β -redex $(\lambda x. a) z$ onde z é um valor ou uma variável.

Big-step semantics

$$(CBV_{bs}) \quad \lambda x. e \Rightarrow_E \lambda x. e$$

$$(CBV_{bs}) \quad \frac{e_1 \Rightarrow_E \lambda x. e \quad e_2 \Rightarrow_E v \quad e[v/x] \Rightarrow_E v'}{e_1 e_2 \Rightarrow_E v'}$$

“Strictness”

- Uma função diz-se *estrita* se diverge sempre que é aplicada a um argumento que diverge.
- A avaliação “call-by-value” é uma estratégia *estrita*, no sentido em que os argumentos das funções são sempre avaliados, independentemente de virem a ser ou não usados no corpo da função.
- Estratégias *não estritas* (ou *preguiçosas*), como é o caso do “call-by-name” e do “call-by-need”, avaliam apenas os argumentos que são efectivamente usados pela função.

Exercício

Apresente a sequência de reduções correspondente à avaliação “call-by-name” e “call-by-value” para as seguintes expressões

- $(\lambda x. x) ((\lambda y. y)(\lambda z. z))$
- $(\lambda x. \lambda y. y) ((\lambda x. x x)(\lambda x. x x))$

Programação directa em λ -calculus

- Apesar da sua definição sucinta, o λ -calculus é muito poderoso.
- Podemos programar directamente em λ -calculus puro desde que a “execução” do programa seja entendida como a redução à forma normal, seguindo a ordem normal de redução.
- A apresentação do λ -calculus puro como uma linguagem de programação serve aqui apenas para ilustrar o seu poder.
- A ideia é codificar os booleanos, números naturais e outros tipos primitivos, com formas normais fechadas apropriadas (funções que imitam o comportamento desses tipos de dados).

Programação directa em λ -calculus

Booleanos

$$\text{TRUE} \equiv \lambda x. \lambda y. x \quad \text{FALSE} \equiv \lambda x. \lambda y. y$$

Com esta codificação, uma expressão condicional “if b then e_1 else e_2 ” pode ser simplesmente escrita como $b e_1 e_2$, dado que

$$\text{TRUE } e_1 e_2 \rightarrow_{\beta}^* e_1 \quad \text{FALSE } e_1 e_2 \rightarrow_{\beta}^* e_2$$

Uma possível definição de operadores

$$\text{NOT} \equiv \lambda b. \lambda x. \lambda y. b y x$$

$$\text{AND} \equiv \lambda b. \lambda c. \lambda x. \lambda y. b (c x y) y$$

Exercício

Mostre que

- $\text{NOT TRUE} \rightarrow_{\beta}^* \text{FALSE}$ e $\text{NOT FALSE} \rightarrow_{\beta}^* \text{TRUE}$
- $\text{AND FALSE TRUE} \rightarrow_{\beta}^* \text{FALSE}$, etc.

Programação directa em λ -calculus

Exercício

Definições alternativas para NOT e AND poderão ser:

$$\text{NOT} \equiv \lambda b. b \text{ FALSE TRUE}$$

$$\text{AND} \equiv \lambda b_1. \lambda b_2. b_1 b_2 \text{ FALSE}$$

- Mostre que estes operadores têm o comportamento adequado.
- Proponha uma definição adequada para o operador OR.

Programação directa em λ -calculus

Numerais de Church

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

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

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

...

Exemplos de uma função e de um predicado

$$\text{SUCC} \equiv \lambda n. \lambda f. \lambda x. f (n f x)$$

$$\text{ISZERO} \equiv \lambda n. \lambda x. \lambda y. n (\lambda z. y) x$$

Exercício

Mostre que

- $\text{SUCC } \bar{n} \rightarrow_{\beta}^* \overline{n+1}$
- $\text{ISZERO } \bar{0} \rightarrow_{\beta}^* \text{TRUE}$ e $\text{ISZERO } \overline{n+1} \rightarrow_{\beta}^* \text{FALSE}$

Programação directa em λ -calculus

Uma codificação da adição e da multiplicação de numerais de Church

$$\text{ADD} \equiv \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)$$

$$\text{MULT} \equiv \lambda m. \lambda n. \lambda f. m (n f)$$

Exercício

Mostre que

- $\text{ADD } \bar{2} \bar{3} \rightarrow_{\beta}^* \bar{5}$
- $\text{MULT } \bar{2} \bar{3} \rightarrow_{\beta}^* \bar{6}$

Programação directa em λ -calculus

Recursão e combinadores de ponto fixo

- A definição de funções recursivas pode ser feita com o auxílio de um combinador de ponto fixo.
- O combinador Y é um dos mais simples e foi descoberto por Haskell Curry.

$$Y \equiv \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

- A ideia é que $Y G$ é ponto fixo da funcional G . Logo,

$$G (Y G) = Y G$$

Programação directa em λ -calculus

Vejamos o exemplo da função factorial

$$\text{FACT } n = \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{FACT}(n - 1)$$

Abusando da notação podemos escrever

$$\text{FACT} = \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{FACT}(n - 1)$$

$$G = \lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f(n - 1))$$

$$\text{FACT} \equiv Y G$$

Com base nas definições apresentadas mostre que

$$\text{FACT} =_{\beta} \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{FACT}(n - 1)$$

Programação directa em λ -calculus

- Para além de não ser prático programar em λ -calculus puro, as codificações de dados podem revelar propriedades que não são desejadas do ponto de vista das entidades que queremos representar.

Por exemplo:

- ▶ $\bar{0} = \text{FALSE}$
- ▶ Uma codificação alternativa da função sucessor

$$\text{SUCC}' \equiv \lambda n. \lambda f. \lambda x. n f (f x)$$

tem um comportamento diferente de SUCC quando aplicada a expressões que não representam números naturais.

- Estes problemas podem ser evitados programando num λ -calculus tipificado.