CÁLCULO LAMBDA: UNINDO COMPUTAÇÃO E LÓGICA

Rodrigo Machado

28 de Novembro de 2017

ALGUMAS PERGUNTAS INICIAIS

- Qual modelo de computação universal é mais conhecido em Ciência da Computação?
- Quem sabe quem foi Alonzo Church?
- Quem sabe o que quer dizer correspondência Curry-Howard?
- Quem já utilizou um assistente de prova?
- Quem programa em Haskell ou Ocaml?
- Qual a grande novidade do Java 8?
- O que é inferência de tipos?
- Qual o aspecto comum a todas as respostas acima?
- Por que um tutorial em 2017 sobre um tópico anterior à segunda guerra mundial?

OBJETIVOS DESTE TUTORIAL

Apresentar uma visão geral sobre diversas versões do *Cálculo Lambda*, e alguns pontos importantes de sua história.

Detalhar o cálculo lambda como:

- modelo de computação
- lógica

Motivar a aplicabilidade dos conceitos formais por meio de duas vertentes: linguagens de programação e sistemas de prova.

AGENDA

1. Cálculo Lambda Sem Tipos
   - Sintaxe
   - Semântica
   - Propriedades
2. Cálculo Lambda como Modelo de Computação
   - Operações lógicas
   - Operações aritméticas
   - Pares ordenados
   - Pontos fixos e recursão
3. Cálculo Lambda Tipado
   - Tipos simples
   - Polimorfismo
   - Tipos parametrizados
   - Tipos dependentes
4. Cálculo Lambda como Lógica
   - Proposições como Tipos
   - Codificações lógicas
5. Aplicações
   - Linguagens Funcionais
   - Assistentes de Prova
6. Conclusões

CÁLCULO LAMBDA

O **cálculo lambda** foi proposto na década de 30 por Alonzo Church.

Um cálculo de **funções**, formalizando essencialmente *definição* e *aplicação* das mesmas.

Church e seus alunos Stephen Cole Kleene e John Barkley Rosser mostraram que esse formalismo é tão expressivo quanto outros modelos de computação propostos.

Alonzo Church

Kleene

Rosser

CÁLCULO LAMBDA: ORIGEM

Considere a seguinte definição matemática:

$$f(x) = x^2 + 7$$

A igualdade acima está *definindo uma função* f, que consome um número e devolve um número.

$$f(3) = 3^2 + 7 = 16$$

A igualdade acima está *aplicando* a definição de f a um valor específico.

O propósito original da **notação lambda** foi resolver a seguinte ambiguidade:

$$f(e) = e^2 + 7$$

Afinal, se trata da aplicação de f sobre a constante $e = 2.716\ldots$ ou da definição de f?

CÁLCULO LAMBDA: ORIGEM (2)

A **notação lambda** permite diferenciar claramente a *definição de uma função* de sua respectiva *aplicação*.

$$f = \lambda x.x^2 + 7$$

Acima temos a definição da função. O λx indica que x deve ser interpretado como um *parâmetro formal*, isto é, um nome temporário para o valor a ser recebido pela função.

$$f(3) = (\lambda x.x^2 + 7)(3) = 3^2 + 7$$

Acima temos a aplicação da função f, definida anteriormente, ao valor 3.

Note que o **significado** da aplicação é a **substituição** do *parâmetro formal* x pelo *valor concreto* 3.

SINTAXE FORMAL DE PRÉ-TERMOS

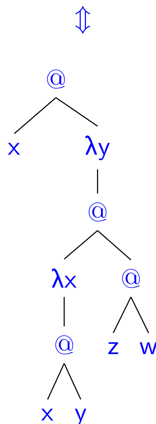
Sintaxe (pré-termos)

$$
\begin{array}{lcl}
x, y, z & \in & \mathsf{Var} \\
M, N & \in & \Lambda^{-} \\
M, N & ::= & x \mid M\ N \mid \lambda x.M
\end{array}
$$

Exemplos:

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

Nota: cada termo é uma árvore de sintaxe



VARIÁVEIS LIVRES, LIGADAS E SUBSTITUIÇÃO

Dentro do pré-termo $\lambda x.M$, dizemos que M é o **escopo** de λx. Uma ocorrência de x dentro do escopo de λx é dita **ligada**. Caso contrário, x ocorre **livre**.

Exemplo: no termo $\lambda a.a\ b$, temos que a ocorre ligada, e b ocorre livre

A notação $M\ [x \leftarrow N]$ descreve o termo resultante da substituição de todas as **ocorrências livres** de x em M por N, **evitando captura de variáveis livres**.

Exemplo: o resultado da substituição $(\lambda a.a\ b)[b \leftarrow (z\ z)]$ é $\lambda a.a\ (z\ z)$

EQUIVALÊNCIA ALFA

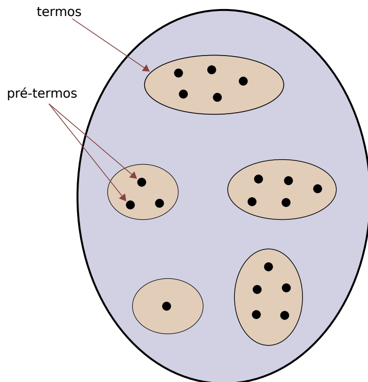
Dois pré-termos M e N são ditos α-equivalentes ($M =_\alpha N$) sss eles diferem somente na escolha dos nomes de variáveis ligadas.

Exemplo: $\lambda x.x\ y =_\alpha \lambda z.z\ y$ e $\lambda x.x\ y \neq_\alpha \lambda y.y\ y$

Um **termo lambda** é uma *classe de equivalência* de pré-termos α-equivalentes.

Exemplo: o termo identidade é $\{\ \lambda x.x,\ \lambda y.y,\ \lambda z.z, \ldots\}$

Definição: $\Lambda = \Lambda^{-}/{=_\alpha}$



REDEXES E FORMAS NORMAIS

Um **redex** *(reducible expression)* é um subtermo de M com o formato

$$(\lambda x.P)\ Q$$

e o seu respectivo **contractum** é

$$P[x \leftarrow Q]$$

Um termo que não contenha nenhum redex é chamado **forma normal** (ou termo irredutível).

Exemplo:

a) $\lambda y.(\lambda x.y\ x)\ ((\lambda y.x)\ y)$ contém dois redexes
b) $\lambda x.\lambda y.x\ x$ é uma forma normal

REDUÇÃO BETA

Redução beta descreve a avaliação de termos lambda através de substituição.

Dizemos que M beta-reduz para N, denotado $M \to_\beta N$, se obtivermos N a partir da contração de algum redex de M.

Exemplo:

$$(\lambda z.\lambda x.x\ z)\ y \ \to_\beta \ \lambda x.x\ y$$

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

Denotamos $M \twoheadrightarrow_\beta N$ quando M reduz para N em 0 ou mais passos de redução beta (é o fecho transitivo e reflexivo da redução beta).

EQUIVALÊNCIA BETA

Equivalência beta identifica termos que possuem reescritas confluentes:

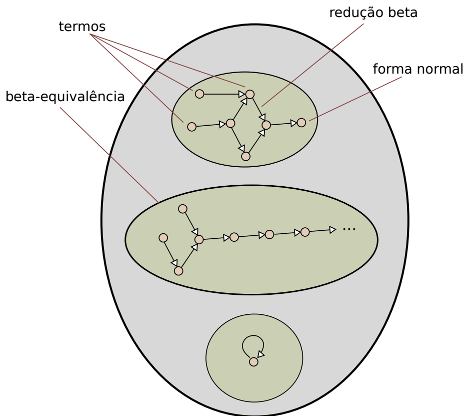
$$\frac{M \twoheadrightarrow_\beta P \qquad N \twoheadrightarrow_\beta P}{M =_\beta N}$$

Dizemos que um termo M **possui forma normal** sss $M =_\beta N$ e N **é uma forma normal**.

Exemplo:

$(\lambda x.x\ x)\ a$ possui forma normal

$\mathbf{\Omega} = (\lambda x.x\ x)\ (\lambda x.x\ x)$ não possui forma normal (ele β-reduz para si mesmo)



ESTRATÉGIAS DE AVALIAÇÃO

Um termo P pode ter diversos redexes e, portanto, avaliar para distintos termos. Uma **estratégia de avaliação** é uma escolha fixa para todos os termos de qual redex tem prioridade na redução.

Avaliação normal:

- mais à esquerda, mais externo
- realiza a aplicação sem normalizar a função ou os argumentos

Avaliação aplicativa (estrita):

- mais à esquerda, mais interno
- normaliza a função e todos os argumentos antes de realizar a aplicação

Exemplo:

- normal: $((\lambda x\ y.x)\ \mathbf{I}\ \mathbf{\Omega}) \rightarrow_\beta (\lambda y.\mathbf{I})\ \mathbf{\Omega} \rightarrow_\beta \mathbf{I}$
- aplicativa: $((\lambda x\ y.x)\ \mathbf{I}\ \mathbf{\Omega}) \rightarrow_\beta (\lambda y.\mathbf{I})\ \mathbf{\Omega} \rightarrow_\beta (\lambda y.\mathbf{I})\ \mathbf{\Omega} \rightarrow_\beta \ldots$

PROPRIEDADES DO CÁLCULO LAMBDA

Teorema: (Confluência, Church-Rosser) se $N \leftarrow_\beta M \rightarrow_\beta N'$ então existe P tal que $N \twoheadrightarrow_\beta P \twoheadleftarrow_\beta N'$

Teorema: se o termo M possui forma normal N, então avaliar M usando a estratégia normal certamente alcança N.

Nota:

- nem todo termo possui forma normal.
- a avaliação estrita nem sempre alcança a forma normal.
- determinar equivalência beta de termos é um problema indecidível.

CÁLCULO LAMBDA COMO MODELO DE COMPUTAÇÃO UNIVERSAL

A *linguagem de termos lambda* juntamente com a redução beta (e uma estratégia de avaliação fixa) compõe um **modelo de computação**.

Podemos enxergar a seguinte associação:

- termo lambda = estado da máquina
- redução beta = passo de execução da máquina
- formas normais = valores finais (parada da máquina)

Todas as funções Turing-computáveis podem ser descritas em cálculo lambda.

CODIFICAÇÕES

Como a linguagem lambda pura provê somente variáveis, abstração lambda e aplicação de termos a termos, é necessário **codificar** em termos lambda

Dados:

- valores booleanos e operações lógicas
- números, operações aritméticas e operações relacionais
- coleções: pares e listas

Estruturas de controle:

- execução condicional
- definições locais
- recursão

Nota: nessas codificações o foco não é *eficiência*!

BOOLEANOS

Definição: booleanos de Church:

$$\textbf{true} = \lambda x\ y\ .\ x$$
$$\textbf{false} = \lambda x\ y\ .\ y$$

Definição: construção `if-then-else` :

$$\textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c = \lambda a\ b\ c\ .\ a\ b\ c$$

Ideia: aplicar o booleano resultante do teste sobre os valores a serem retornados.

Justifica a definição de valores-verdade como *seletores.*

Definição: numerais de Church:

$$\mathbf{0} = \lambda f\ x\ .\ x$$
$$\mathbf{1} = \lambda f\ x\ .\ f\ x$$
$$\mathbf{2} = \lambda f\ x\ .\ f\ (f\ x)$$
$$\mathbf{3} = \lambda f\ x\ .\ f\ (f\ (f\ x))$$
$$\vdots$$
$$\mathbf{n} = \lambda f\ x\ .\ \overbrace{f\ (f\ (f\ \ldots (f}^{n}\ x) \ldots))$$

Também referenciados por $\mathbf{c_n}$, para $n \in \mathbb{N}$.

OPERAÇÕES ARITMÉTICAS: SUCESSOR

Função sucessor:

$$\textbf{succ} = \lambda n\ p\ q\ .\ p\ (n\ p\ q)$$

Ideia da construção:

1. receber um numeral de Church n com estrutura

$$n = \lambda f\ x\ .\ f\ (f \ldots (f\ x) \ldots)$$

2. o resultado deve ter p e q ligados por lambdas:

$$\lambda p\ q\ .\ \ldots$$

3. o termo (n p q) muda os f's em p's, e o x em q.
4. por último, introduzimos um p adicional ao número.

OPERAÇÕES ARITMÉTICAS E RELACIONAIS

Definição: operações aritméticas básicas:

$$\mathbf{add} = \lambda\ m\ n\ p\ q\ .\ m\ p\ (n\ p\ q)$$
$$\mathbf{mult} = \lambda\ m\ n\ p\ q\ .\ m\ (n\ p)\ q$$
$$\mathbf{exp} = \lambda\ m\ n\ .\ n\ m$$

Definição: teste de zero:

$$\mathbf{isZero} = \lambda n\ .\ n\ (\lambda x.\mathbf{false})\ \mathbf{true}$$

Nota: a função predecessor é trabalhosa e será vista posteriormente.

PARES ORDENADOS

Definição: um par (M, N) pode ser representado por **pair** $M\ N$ onde

$$\textbf{pair} = \lambda\ m\ n\ b.\ b\ m\ n$$

Exemplo:

$$\textbf{pair}\ \mathbf{0}\ \textbf{true} = \lambda b.\ b\ \mathbf{0}\ \textbf{true}$$

Definição: funções de acesso ao conteúdo de um par

$$\textbf{fst} = \lambda p.\ p\ \textbf{true}$$

$$\textbf{snd} = \lambda p.\ p\ \textbf{false}$$

PARES ORDENADOS: PREDECESSOR

Usando pares podemos definir $\mathsf{pred} : \mathbb{N} \to \mathbb{N}$ onde

$$\mathsf{pred}(n) = \begin{cases} 0 & \text{se} \quad n \leq 1 \\ n - 1 & \text{caso contrário} \end{cases}$$

Função auxiliar: $\mathsf{shiftInc}(a, b) = (b, b + 1)$

$$\mathbf{shiftInc} = \lambda p.\mathbf{pair}\ (\mathbf{snd}\ p)\ (\mathbf{succ}\ (\mathbf{snd}\ p))$$

Definição: o termo **pred** abaixo implementa a função pred

$$\mathbf{pred} = \lambda n.\mathbf{fst}\ (n\ \mathbf{shiftInc}\ (\mathbf{pair}\ \mathbf{0}\ \mathbf{0}))$$

PONTOS FIXOS: COMBINADOR Y

Chamamos o termo abaixo de **combinador Y** (ou combinador de ponto fixo)

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

Teorema: o combinador **Y** produz um ponto fixo para seu argumento

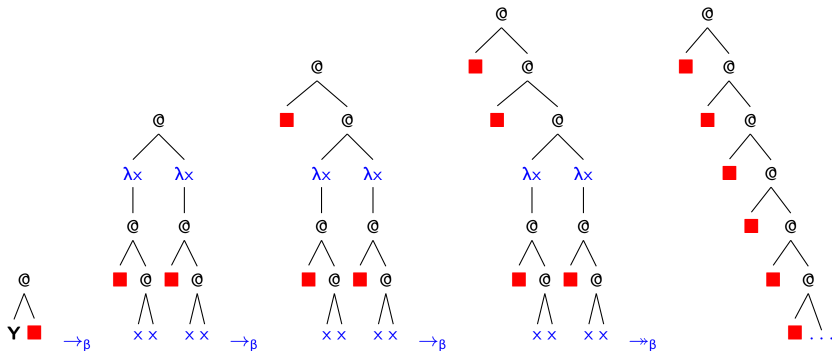
Demonstração:

$$\begin{aligned} \mathbf{Y}S &\to_\beta \\ (\lambda x.S(x\ x))\ (\lambda x.S(x\ x)) &\to_\beta \\ S\ (\lambda x.S(x\ x))\ (\lambda x.S(x\ x)) &=_\beta \\ S(\mathbf{Y}S) & \end{aligned}$$

$\mathbf{Y}S =_\beta S(\mathbf{Y}S)$ e, portanto, $\mathbf{Y}S$ é um ponto fixo de S □

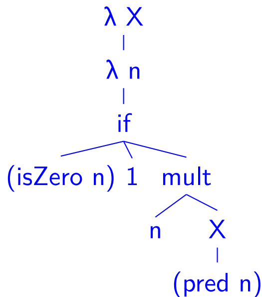
Utilizando o combinador Y, podemos construir cadeias "infinitas" (sob demanda) de expressões if-then-else, simulando recursão.

PONTOS FIXOS: COMBINADOR Y EM AÇÃO



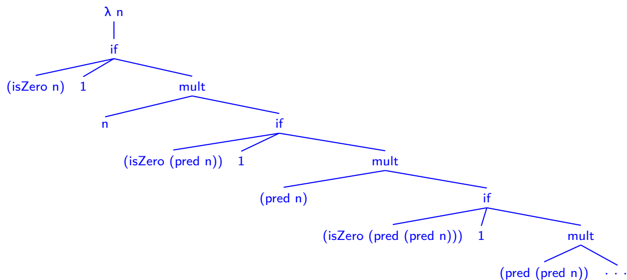
PONTOS FIXOS: COMBINADOR Y EM AÇÃO (2)

se S $=_\beta$



PONTOS FIXOS: COMBINADOR Y EM AÇÃO (3)

então $(\mathbf{Y}\ \mathsf{S}) =_\beta$



FUNÇÕES RECURSIVAS E OPERADOR Y

Usando **Y** podemos codificar funções como termos nos baseando em suas **definições recursivas**.

Definição recursiva:

$$\mathsf{fact}(n) = \begin{cases} 1 & \text{se} \quad n \leq 1 \\ n * \mathsf{fact}(n-1) & \text{caso contrário} \end{cases}$$

Combinador:

$$\begin{aligned} F = \lambda M.\lambda n.\mathbf{if}\ & (\mathbf{isZero}\ (\mathbf{pred}\ n)) \\ & \mathbf{1} \\ & (\mathbf{mult}\ n\ (M\ (\mathbf{pred}\ n))) \end{aligned}$$

Definição final: $\mathbf{fact} = \mathbf{Y}F$

UNIVERSALIDADE

Sendo possível definir:

- booleanos e condicionais
- aritmética natural
- estruturas de dados
- recursão

temos os componentes de uma linguagem de programação que permite representar qualquer algoritmo.

A avaliação de termos lambda pode entrar em laço infinito, visto que nem todo termo possui forma normal.

Motivação: considere

- **if** $= \lambda$a b c.a b c
- **2** $= \lambda$f x.f (f x)
- **3** $= \lambda$f x.f (f (f x))

Pergunta: qual o resultado da avaliação expressão **if 2 2 3** visto que **2** $\neq$ **true** e **2** $\neq$ **false** ?

Resposta: algum termo “estranho” (no caso, **81**)

Em outras palavras não há uma noção de compatibilidade entre funções e argumentos: podemos aplicar qualquer termo a qualquer outro termo (inclusive realizar uma auto-aplicação).

TIPOS SIMPLES

Podemos utilizar **tipos** para categorizar termos da linguagem.

Definição: linguagem de tipos

$$\begin{aligned} \mathsf{T} ::= {} & \alpha, \beta, \gamma, \ldots && \text{tipos básicos} \\ \mid {} & \mathsf{T} \to \mathsf{T} && \text{tipos funcionais} \end{aligned}$$

Exemplo:

$$\alpha \qquad \alpha \to \alpha$$

$$\alpha \to \beta \qquad \alpha \to (\alpha \to \alpha)$$

$$(\alpha \to \alpha) \to (\beta \to \beta) \qquad (\alpha \to \beta) \to \gamma$$

INTERPRETAÇÃO DE TIPOS

Tipos básicos (referenciado por α) podem ser interpretados como tipos primitivos disponíveis em uma dada arquitetura de computador (tais como booleanos ou inteiros)

Tipos funcionais (referenciados por $T_1 \to T_2$) são tratados como funções que recebem elementos do domínio T_1 e geram elementos do contradomínio T_2.

Exemplo:

$$42 : \mathbb{N}$$
$$\lambda x.x + 1 : \mathbb{N} \to \mathbb{N}$$
$$\lambda f.\ \lambda x.\ f(f\ x) : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$$

Notação: a expressão $e : T$ significa "termo e é do tipo T"

AMBIENTES DE TIPOS

Definição: um **ambiente de tipos** Γ é uma sequência de atribuições de tipos para variáveis $x, y, z, \ldots \in \mathsf{Var}$.

Exemplo:

- $\Gamma_1 = \varepsilon$
- $\Gamma_2 = x : \mathsf{Int}, y : \mathsf{Bool}, f : \mathsf{Int} \to \mathsf{Int}$

Ambientes pode ser

- consultados: $\Gamma_2(x) = \mathsf{Int}$
- atualizados: $\Gamma_3 = \Gamma_2, x : \mathsf{Bool}$

JULGAMENTO DE TIPO

Notação: um **julgamento de tipo**

$$\Gamma \vdash e : T$$

é uma afirmação de que o termo e é do tipo T sob ambiente Γ.

Exemplo: $x : \alpha, y : \beta \vdash x : \alpha$

Definição: um **sistema de tipos** é uma coleção de regras de dedução que permitem construir julgamentos de tipos

CÁLCULO LAMBDA TIPADO SIMPLES

Sintaxe:

$$e ::= \; x \mid e\ e' \mid \lambda x : T.e$$

Exemplo: $\lambda x : \alpha . \lambda y : \beta . y$

Sistema de Tipos

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad (\textsc{Var})$$

$$\frac{\Gamma \vdash e : T \to T' \qquad \Gamma \vdash e' : T}{\Gamma \vdash e\ e' : T'} \quad (\textsc{App})$$

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \lambda x : T.e : T \to T'} \quad (\textsc{Lam})$$

SISTEMA DE TIPOS SIMPLES: PROPRIEDADES

O Cálculo Lambda Tipado Simples possui diversas propriedades interessantes:

- **Confluência**: se $e \to_\beta e_1$ e $e \to_\beta e_2$, então existe e' tal que $e_1 \twoheadrightarrow_\beta e'$ e $e_2 \twoheadrightarrow_\beta e'$
- **Preservação de tipos**: se $\Gamma \vdash e : T$ e $e \to_\beta e'$, então $\Gamma \vdash e' : T$
- **Normalização**: se $\varepsilon \vdash e : T$ então existe forma normal e' tal que $e \twoheadrightarrow_\beta e'$

Diversos termos importantes do cálculo lambda sem tipos **não são termos válidos de $\lambda_\to$**. Por exemplo, não há tipo possível para o termo

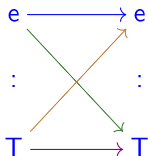
$$\lambda x{:}?.x\ x$$

O Cálculo Lambda Tipado Simples **não é Turing-completo**. **Todos os termos possuem forma normal**.

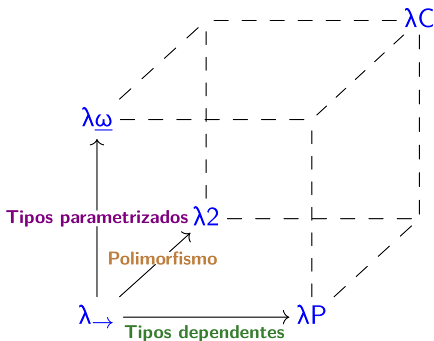
CUBO LAMBDA

O Cálculo Lambda Tipado Simples somente apresenta abstração de termos para termos. Isso pode ser generalizado:

Dependências entre tipos e termos



Cubo lambda



POLIMORFISMO

Slogan: *termos* que dependem de *tipos*.

Polimorfismo = poli (múltiplas) + morfos (formas). Duas interpretações principais:

- uma única definição de rotina aplicável em diversos cenários distintos (polimorfismo paramétrico, “generics”)
- uma coleção de rotinas distintas que compartilham o mesmo nome, sendo a escolha de qual será usada determinada pelo contexto (polimorfismo ad-hoc, overloading)

Exemplo: (em Haskell)

```
ident :: forall a. a -> a
ident x = x
foo1 = 1 + 1
foo2 = 1 + 4.0
```

TIPOS PARAMETRIZADOS

Slogan: *tipos* que dependem de *tipos (construtores de tipos)*

Considere as seguintes definições de listas em Haskell.

```
data ListaInt  = Nempty
               | Ncons Int  ListaInt
l1 = Ncons 3 (Ncons 5 Nempty)
```

```
data ListaChar = Cempty
               | Ccons Char ListaChar
l2 = Ccons 'a' (Ccons 'b' (Ccons 'c' Cempty ))
```

Note que os dois tipos são similares: onde o único ponto de variação é o tipo do dado armazenado na lista.

TIPOS PARAMETRIZADOS

Slogan: *tipos* que dependem de *tipos (construtores de tipos)*

Possibilidade: abstrair o tipo de listas como um construtor de tipos, isto é, uma função $\texttt{List} : \mathsf{Type} \to \mathsf{Type}$

```
1  data List a = Empty
2               | Cons a (List a)
3
4  l3 = Cons 4 (Cons 2 Empty)
```

Nota: em Haskell, tipos básicos são referenciados por $\star$. Expressões formadas por $\star$ e $\to$ são chamadas "kinds" (ou "tipos dos tipos")

Exemplo: $\texttt{List} : \star \to \star$

TIPOS DEPENDENTES

Slogan: *tipos* que dependem de *termos* (famílias de tipos indexadas por valores).

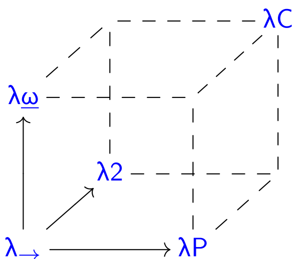
Exemplo: listas com tamanho fixo (vec) em Coq.

```
1 Inductive vec : nat -> Type :=
2   | nil  : vec 0
3   | cons : forall (n:nat),
4              nat -> vec n -> vec (S n).
```

Nessa linguagem podemos criar versões de operações como **indexação**, **soma elemento-a-elemento** e **concatenação** que carregam o tamanho da estrutura como parte do tipo.

Cálculo de construções: λC

$\lambda C = \lambda_{\rightarrow} + \lambda 2 + \lambda \underline{\omega} + \lambda P$



O cálculo λC inclui polimorfismo, construtores de tipos e tipos dependentes.

Apesar desta extensões, mantém características importantes de $\lambda_{\rightarrow}$ como **normalização forte** e **confluência**.

Muitos termos que não são bem-tipados em $\lambda_{\rightarrow}$ podem ter tipos atribuídos em λC.

Exemplo: auto-aplicação $\lambda a.a\ a$

Sintaxe de λC

A, B	$::=$	$\square$	(tipo de todos os *kinds*)
	$\mid$	$\star$	(tipo dos *tipos simples*)
	$\mid$	x	(variáveis)
	$\mid$	$A\ B$	(aplicação)
	$\mid$	$\lambda x : A.B$	(dependência funcional)
	$\mid$	$\Pi x : A.B$	(produto dependente)

$$K \text{ é um kind} \Leftrightarrow \vdash K : \square$$
$$T \text{ é um tipo} \Leftrightarrow \vdash T : K : \square$$
$$E \text{ é um termo} \Leftrightarrow \vdash E : T : K : \square$$

Açúcar sintático:

- $A \to B$ é açúcar sintático para $\Pi x : A.B$ para $x \notin FV(B)$.
- $\forall T.B$ é açúcar sintático para $\Pi T : \star.B$

SISTEMA DE TIPOS

Conformidade: $\Gamma \vdash A$ (todas as variáveis em A estão declaradas em Γ e são consistentes)

$(s_1, s_2) \in \{(\star, \star), (\star, \Box), (\Box, \star), (\Box, \Box)\}$

Sistema de tipos

$$\varepsilon \vdash \star : \Box \quad (\textsc{sort})$$

$$\frac{\Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A \vdash x : A} \quad (\textsc{var})$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s \qquad x \notin \Gamma}{\Gamma, x : C \vdash A : B} \quad (\textsc{weak})$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2} \quad (\textsc{form})$$

$$\frac{\Gamma \vdash M : \Pi x{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[x \leftarrow N]} \quad (\textsc{appl})$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B} \quad (\textsc{abst})$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'} \quad (\textsc{conv})$$

Semântica

$$(\lambda x : A.B)\ C \to_{\beta} B[x \leftarrow C] \qquad (\textsc{Beta})$$

$$(\Pi x : A.B)\ C \to_{\beta} B[x \leftarrow C] \qquad (\textsc{BetaPi})$$

Nota: distinção entre termos e tipos exclusivamente por conta do sistema de tipos (mesma linguagem)

LÓGICA E CÁLCULO LAMBDA

Um similaridade interessante surge quando comparamos regras de lógica e regras de tipos do cálculo lambda tipado simples:

Exemplo: Modus Ponens

$$\frac{A \to B \qquad A}{B} \quad (\mathrm{MP})$$

Exemplo: Regra da aplicação

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f\ x : B} \quad (\mathrm{APP})$$

A única diferença está na forma como registramos as subprovas (árvores de dedução vs ambientes e termos).

PROPOSIÇÕES COMO TIPOS

Cada versão do cálculo lambda tipado corresponde a um sistema dedutivo de uma lógica (intuicionista).

- $\lambda_{\rightarrow}$: lógica proposicional minimal
- $\lambda 2$: lógica proposicional de segunda ordem
- λP : lógica de predicados de primeira ordem minimal

Um cálculo lambda tipado pode ser considerado tanto um modelo de computação quanto uma lógica.

A conexão entre essa visões é denominada *Correspondência Curry-Howard* ou *Proposições como Tipos*, na qual

- tipos da linguagem são vistos como fórmulas lógicas (proposições)
- termos (bem-tipados) são vistos como provas do respectivo tipo

HABITAÇÃO DE TIPOS

Um tipo T é **habitado** se existe termo e tal que $\vdash e : T$.

A seguinte interpretação é utilizada:

- se um tipo é habitado, ele possui uma prova. Portanto, a proposição é verdadeira.
- se um tipo não é habitado, ele não possui prova. Portanto, a proposição é falsa.

Exemplo: em λC

- $\Pi X{:}\star.X \to X$ é habitado pelo termo $\lambda X{:}\star.\lambda x{:}X.x$
- $\Pi X{:}\star.X$ não é habitado, pois não conseguimos construir um expressão que escolhe um termo a partir de um tipo qualquer.

Nota: podemos utilizar $\Pi X{:}\star.X$ como $\bot$ (falso), e $\Pi X{:}\star.X \to X$ como $\top$ (true).

CODIFICAÇÃO DE OPERADORES LÓGICOS

No cálculo λC, podemos codificar lógica de predicados intuicionista:

- **Implicação** $A \supset B$ é representada diretamente pelo tipo funcional: $A \to B$ ou ($\Pi_ : A.B$ na sintaxe de λC)
- **Negação** $\neg A$ pode ser codificada por $A \to \Pi X{:}\star.X$ visto que $\neg A \equiv A \supset \bot$
- **Conjunção** $A \wedge B$ é representada por $\Pi C{:}\star.(A \to B \to C) \to C$
- **Disjunção** $A \vee B$ é representada por $\Pi C{:}\star.(A \to C) \to (B \to C) \to C$
- **Quantificação universal** como $\forall x \in S,\ P(x)$ é representada diretamente utilizando o tipo dependente $\Pi x : S.(P\ x)$
- **Quantificação existencial** como $\exists x \in S,\ P(x)$ é representada diretamente utilizando o tipo dependente $\Pi Y : \star.(\Pi x : X.P\ x \to Y) \to Y$ (note que $x \notin FV(Y)$)

APLICAÇÕES

Pergunta: Ok. Interessante. O que eu faço com isso?

Resposta: no mínimo as seguintes coisas bastante importantes. . .

- programação de computadores
- estudo formal de linguagens de programação
- matemática em geral

Vou mencionar dois exemplos importantes:

- Linguagem de Programação (Ex: Haskell)
- Assistente de Provas (Ex: Coq)

LINGUAGENS FUNCIONAIS

Não há definição exata do que é uma linguagem funcional, porém acredito que uma boa aproximação seria:

> *Uma linguagem que favorece e encoraja a definição de funções e a composição das mesmas como princípio fundamental para construção de programas.*

Linguagens funcionais podem ser

- puras (Haskell) ou impuras (OCAML, Scala, LISP)
- dinamicamente tipadas (LISP, Scheme) ou estaticamente tipadas (OCAML, Haskell)

LINGUAGENS FUNCIONAIS E CÁLCULO LAMBDA

Linguagens de programação estaticamente tipadas são essencialmente

- cálculo lambda com constantes
- sistema de tipos
- estratégia de avaliação
- operador de recursão (como primitiva)

Atualmente a maior parte das linguagens de programação de uso geral é **híbrida**, suportando o **estilo de programação funcional**. Exemplo: Python, Javascript, Ruby.

Suporte a *closures* (termos lambda em contexto léxico) constituem novidade importante em linguagens já estabelecidas (por exemplo, Java)

LINGUAGEM HASKELL

Haskell é uma linguagem de uso geral funcional pura e com avaliação tardia.

O sistema de tipos de Haskell suporta polimorfismo, construtores de tipo e inferência de tipos.

Haskell não é a mais popular das linguagens de programação existentes, contudo ela é extremamente influente por ter disseminado conceitos importantes como

- avaliação tardia (lazyness)
- programação funcional pura (sem efeitos colaterais)
- mônadas como forma de estruturar código
- typeclasses para integrar polimorfismo paramétrico e polimorfismo ad-hoc

LINGUAGEM HASKELL: EXEMPLO DE CÓDIGO

Exemplo: Quicksort em Haskell

```haskell
quicksort :: (Ord a) => [a] -> [a]

quicksort [] = []

quicksort (x:xs) =
 let
     smallSorted = quicksort [a | a <- xs, a <= x]
     bigSorted   = quicksort [a | a <- xs, a > x]
 in
     smallSorted ++ [x] ++ bigSorted
```

ASSISTENTES DE PROVA

Um **assistente de prova** é uma ferramenta que auxilia determinação de propriedade e construção de provas para as mesmas, garantindo a sua validade.

Eles diferem de **provadores automatizados de teoremas** ou **verificadores de modelos**, pois assistentes requerem intervenção humana.

Exemplo:

- Coq
- Agda
- Isabelle
- L∃∀N

COQ

Coq é uma assistente de provas construído sobre uma extensão de λC (adicionando tipos indutivos como primitivas), mantendo propriedades importantes como consistência e normalização.

Funciona via Curry-Howard:

- programas $\Leftrightarrow$ provas
- tipos $\Leftrightarrow$ propriedades

Coq permite:

- programar (restrito a um subconjunto de funções recursivas)
- definir objetos matemáticos (conjuntos, listas, relações)
- especificar propriedades (sobre programas e objetos matemáticos)
- provar propriedades (usando uma linguagem de tática)

EXEMPLO DE CÓDIGO

Definição de uma lista de naturais, e uma operação de concatenação.

```
1  Inductive natlist :=
2   | empty    : natlist
3   | prefix   : nat -> natlist -> natlist.
4
5  Example ex1 : natlist := prefix 4 (prefix 2 empty).
6
7  Fixpoint append (n1 n2:natlist) : natlist :=
8  match n1 with
9   | empty          => n2
10  | prefix h t     => prefix h (append t n2)
11 end.
12
13 Notation " A |+| B " := (append A B)
14                  (at level 50, left associativity).
```

EXEMPLO DE PROVA SOBRE OPERAÇÕES

Definição de uma propriedade de `natlists`, e sua respectiva prova utilizando a linguagem de prova.

```
1  Theorem app_assoc :
2
3     forall (n1 n2 n3:natlist),
4        (n1 |+| n2) |+| n3 = n1 |+| (n2 |+| n3).
5
6  Proof.
7  induction n1.
8    (* n1 empty *)
9    intros. simpl. reflexivity.
10   (* n1 prefix *)
11   intros. simpl. rewrite IHn1. reflexivity.
12 Qed.
```

EXEMPLO DE PROVA EM DEDUÇÃO NATURAL

Considere o seguinte sequente em dedução natural.

$$A \to B, A \to C, (B \wedge C) \to D \vdash A \to D$$

A árvore de prova da propriedade acima pode ser vista abaixo:

$$\dfrac{\dfrac{\dfrac{\dfrac{A^1 \quad A \to B}{B}\,(\to_e) \quad \dfrac{A^1 \quad A \to C}{C}\,(\to_e)}{B \wedge C}\,(\wedge_i) \quad (B \wedge C) \to D}{D}\,(\to_e)}{A \to D}\,(\to_i \ 1)$$

EXEMPLO DE PROVA EM DEDUÇÃO NATURAL (CONT.)

O seguinte script Coq constrói a árvore de prova do seguinte sequente:
$\vdash (A \to B) \to (A \to C) \to (B \land C \to D) \to A \to D$

```
1  Variables A B C D:Prop.
2  Theorem thm1 : (A->B) -> (A->C) -> ((B/\C)->D) ->
   ↪  (A->D).
3  Proof.
4  intros P1 P2 P3.
5  intro HA.
6  assert B. apply P1. assumption.
7  assert C. apply P2. assumption.
8  assert (B/\C). apply conj. assumption. assumption.
9  apply P3. assumption.
10 Qed.
```

EXEMPLOS MAIS SIGNIFICATIVOS

Compcert C compiler

- compilador C escrito em Coq
- performance do código gerado comparável à do código gerado pelo GCC
- passos de tradução provados corretos
- http://compcert.inria.fr/compcert-C.html

seL4 (Secure Embedded L4 microkernel)

- formalização de um microkernel de sistema operacional em Isabelle (outro assistente de prova baseado em lógica de alta ordem)
- ver http://ssrg.nicta.com.au/

CONCLUSÕES

Cálculo lambda está presente na área da computação desde o princípio, a aproximadamente 80 anos.

Apesar de não ser popular como o modelo Máquina de Turing, é muito influente e relevante em áreas como

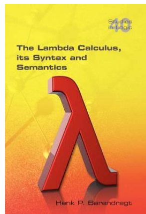
- linguagens de programação
- sistemas de tipos
- assistentes de prova

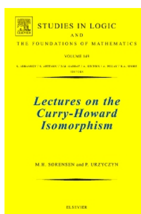
Estudar cálculo lambda assegura uma melhor compreensão da relação entre computação e lógica, assim como ferramentas que habitam essa região (como modernos assistentes de prova).

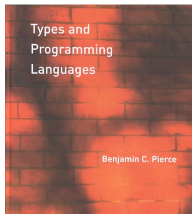
REFERÊNCIAS

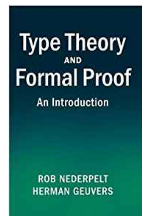
Para saber mais:

- programe em linguagens funcionais (ou em estilo funcional)
- use assistentes de prova ao investigar matemática
- referências:









rma@inf.ufrgs.br

Rodrigo Machado
Instituto de Informática — UFRGS