Cálculo Lambda

Cristiano Damiani Vasconcellos

cristiano.vasconcellos@udesc.br

Departamento de Ciência da Computação
Universidade do Estado de Santa Catarina

Cálculo lambda (λ-cálculo) é um modelo matemático capaz de ilustrar de forma simples alguns importantes conceitos presentes em linguagens de programação, por exemplo, ligação, escopo, ordem de avaliação, computabilidade, sistemas de tipos, etc. O Cálculo lambda consiste em definições de funções e regras reescrita (substituição de variáveis) e foi definido na década de 30 por Alonzo Church com a finalidade de formalizar o conceito de computabilidade.

Uma expressão lambda (em sua forma pura) é definida pela seguinte sintaxe:

`<expressão>`	$\rightarrow$ x	(variável)
	| `<expressão><expressão>`	(aplicação)
	| λx.`<expressão>`	(abstração lambda ou função)

Variáveis Livre e Ligadas

A ocorrência de uma variável, em uma λ-expressão, pode ser *livre* ou *ligada*. Uma variável é *livre* se na expressão não esta associada a nenhuma λ-abstração, por exemplo, a variável x na expressão $x + 1$ e a variável y na expressão $\lambda x.x + y$. Uma variável que não é *livre* é dita *ligada*. Expressões que diferem apenas pelos nomes das variáveis ligadas são chamadas de expressões α-equivalentes e representam exatamente a mesma expressão (sinônimos), por exemplo:

$\lambda x.x$ define a mesma função (identidade) que $\lambda y.y$

O conjunto de variáveis livres em uma expressão M é representado por $FV(M)$ (*free variables*) que é definido como:

$$
\begin{array}{ll}
FV(x) & = \{x\} \\
FV(MN) & = FV(M) \cup FV(N) \\
FV(\lambda x.M) & = FV(M) - \{x\}
\end{array}
$$

Uma expressão M é dita fechada (um termo fechado também é chamado de **combinador**) se $FV(M) = \emptyset$.

Substituição e Equivalência

A substituição de todas ocorrências uma variável x por uma expressão M é representada por $[M/x]$. Podemos obter uma expressão α-equivalente a $\lambda x.M$ substituindo todas as ocorrências de x por y, desde que y não ocorra em M, essa substituição é representada por: $\lambda y.[y/x]M$.

O axioma central de λ-cálculo envolve substituições e é chamado de β-equivalência. A β-equivalência representa a aplicação de uma λ-abstração (função) em um argumento:

$$(\lambda x.M)N = [N/x]M$$

Por exemplo: $(\lambda f.fx)(\lambda y.y)$ é equivalente à $(\lambda y.y)x$ e equivalente à x.

Escrevemos $M \rightarrow N$, M é reduzida a N, quando uma expressão M pode ser reduzida a uma expressão N, β-equivalente, aplicando uma substituição em M. Geralmente esse processo pode ser repetido várias vezes, uma expressão na qual nenhuma β-redução pode ser aplicada está na **forma normal**.

Ordem de Avaliação

As regras de avaliação não especificam a ordem exata que uma expressão deve ser reduzida, uma possível ordem de avaliação é reduzir completamente o argumento antes de substituí-lo no corpo da função, essa avaliação e chamada **avaliação por valor** (*call-by-value* ou *eager evaluation*). Uma outra alternativa é avaliar o argumento apenas se necessário, essa ordem de avaliação e chamada de **avaliação preguiçosa** (*lazy evaluation*, *call-by-need* ou *normal order*[1])

[1]Existe uma pequena diferença no significado.

Avaliação por Valor:

$(\lambda w \lambda y \lambda x.y(wyx))((\lambda a \lambda b \lambda c.b(abc))(\lambda s \lambda z.z))$
$\rightarrow (\lambda w \lambda y \lambda x.y(wyx))(\lambda b \lambda c.b((\lambda s \lambda z.z)bc)))$
$\rightarrow (\lambda w \lambda y \lambda x.y(wyx))(\lambda b \lambda c.b((\lambda z.z)c))$
$\rightarrow (\lambda w \lambda y \lambda x.y(wyx))(\lambda b \lambda c.b(c))$
$\rightarrow \lambda y \lambda x.y((\lambda b \lambda c.b(c))yx)$
$\rightarrow \lambda y \lambda x.y(\lambda c.y(c)x)$
$\rightarrow \lambda y \lambda x.y(y(x))$

Avaliação Preguiçosa:

$(\lambda w \lambda y \lambda x.y(wyx))((\lambda a \lambda b \lambda c.b(abc))(\lambda s \lambda z.z))$
$\rightarrow \lambda y \lambda x.y(((\lambda a \lambda b \lambda c.b(abc))(\lambda s \lambda z.z))yx)$
$\rightarrow \lambda y \lambda x.y((\lambda b \lambda c.b((\lambda s \lambda z.z)bc))yx)$
$\rightarrow \lambda y \lambda x.y((\lambda b \lambda c.b((\lambda z.z)c))yx)$
$\rightarrow \lambda y \lambda x.y((\lambda b \lambda c.b(c))yx)$
$\rightarrow \lambda y \lambda x.y((\lambda c.y(c))x)$
$\rightarrow \lambda y \lambda x.y(y(x))$

Currificação (Currying)

Podemos representar uma função que recebe dois argumentos como $\lambda x.(\lambda y.M)$, sendo M é uma expressão lambda, possivelmente envolvendo x e y. Aplicando um único argumento a essa função temos como retorno uma função que aceita o segundo argumento y. Por exemplo, a função matemática $f(g, x) = g(x)$ tem dois argumentos, mas poderia ser representada em λ-cálculo como:

$$f_c = \lambda g \lambda x.gx$$

A diferença entre f e f_c é que a função f deve receber um par de argumentos (g, x) enquanto f_c pode receber um único argumento g, retornando nesse caso $\lambda x.gx$. Depois de passados todos os argumentos para a função f_c o resultado é exatamente o mesmo da função f. Funções como f_c passaram a ser conhecidas como *funções Currificadas* depois que o matemático Haskell Curry estudou suas propriedades.

Numerais de Church

Os números naturais podem ser expressos por meio do *zero* e seus sucessores, dessa forma o número 1 é representado por *suc(zero)*, o número 2 é representado por *suc(suc(zero))* e assim sucessivamente. Uma possível representação dos números naturais em λ-cálculo é representar o *zero* como a λ-abstração $\lambda s \lambda z.z$ e seus sucessores como:

$$
\begin{aligned}
&1 \equiv \lambda s \lambda z.s(z)\\
&2 \equiv \lambda s \lambda z.s(s(z))\\
&3 \equiv \lambda s \lambda z.s(s(s(z)))\\
&\ldots
\end{aligned}
$$

Considerando essa representação para os números naturais a função para calcular o sucessor de um número qualquer é definida como:

$$SUC = \lambda w \lambda y \lambda x.y(wyx)$$

Por exemplo, o sucessor de *zero* pode ser obtido aplicando a representação de *zero* na função SUC:

$(\lambda w \lambda y \lambda x.y(wyx))(\lambda s \lambda z.z)$
$\rightarrow \lambda y \lambda x.y((\lambda s \lambda z.z)yx)$
$\rightarrow \lambda y \lambda x.y((\lambda z.z)x)$
$\rightarrow \lambda y \lambda x.y(x)$

o resultado é α-equivalente a representação de 1.

Booleanos

Os valores booleanos *verdade* e *falsidade* podem ser representados pelas λ-abstrações:

$$V \equiv \lambda x \lambda y.x$$
$$F \equiv \lambda x \lambda y.y$$

E as operações lógicas definidas como:

$$\begin{aligned} E &= \lambda x \lambda y.xy(\lambda u \lambda v.v) \\ OU &= \lambda x \lambda y.(x(\lambda u \lambda v.u))y \\ NAO &= \lambda x.(x(\lambda u \lambda v.v))(\lambda a.\lambda b.a) \end{aligned}$$

Embora qualquer computação possa ser expressada por meio de λ-cálculo puro, o λ-cálculo enriquecido (com a expressão condicional, algarismos arábicos, operadores aritméticos, operadores relacionais, operadores lógicos, etc) é comumente usado, uma vez que a manipulação de expressões em λ-cálculo enriquecido é consideravelmente mais simples do que em sua versão pura:

$\lambda x.x + 1$	(função incremento)
$\lambda x \lambda y.\texttt{if } x > y \texttt{ then } x \texttt{ else } y$	(função que retorna o maior entre dois elementos)

Operador de Ponto Fixo (FIX)

Uma função recursiva possui uma ou mais referências a si mesma na própria declaração. Por exemplo, a função que calcula o fatorial de um número natural:

$$\mathit{fat} = \lambda n.\texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } n * \mathit{fat}(n-1)$$

Como expressar recursão em funções anônimas?

Considere a função G obtida através de uma pequena alteração na definição acima, tornando a função um argumento:

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

Operador de Ponto Fixo (FIX)

O ponto fixo da função G é um valor f tal que $G(f) = f$. Funções recursivas podem ser expressas, em λ-cálculo, usando o operador de ponto fixo (FIX). O operador de ponto fixo é uma expressão tal que para toda expressão F:

$$FIX\ \ F = F\ \ (FIX\ \ F)$$

Usando o operador de ponto fixo definimos a função fatorial da seguinte maneira:

$$fat \texttt{ = } FIX\ \ \lambda f \lambda n.\texttt{if } n \texttt{ = 0 then 1 else } n \texttt{ * } f\texttt{(}n\texttt{-1)}$$

Operador de Ponto Fixo (FIX)

Cada vez operador de ponto fixo é reduzido *FIX F* deve ser substituído por *F FIX F*. Como no exemplo abaixo no qual é calculado o fatorial de 1.

$(FIX\ \lambda f \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f\ (n-1))\ 1$
$\rightarrow((\lambda f \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f\ (n-1))$
$(FIX\ \lambda f \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f\ (n-1)))\ 1$
$\rightarrow(\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n *$
$(FIX\ \lambda f \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f\ (n-1))\ (n-1))\ 1$
$\rightarrow\texttt{if}\ 1 = 0\ \texttt{then}\ 1\ \texttt{else}\ 1 *$
$((FIX\ \lambda f \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f\ (n-1))\ 0)$
$\rightarrow\texttt{if}\ 1 = 0\ \texttt{then}\ 1\ \texttt{else}\ 1 *$
$((\lambda f \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f\ (n-1))$
$(FIX\ \lambda f \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f\ (n-1))\ 0)$
$\rightarrow\texttt{if}\ 1 = 0\ \texttt{then}\ 1\ \texttt{else}\ 1 * ((\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n *$
$(FIX\ \lambda f \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f\ (n-1)))\ 0)$
$\rightarrow\ 1*1$
$\rightarrow\ 1$

Operador de Ponto Fixo (FIX)

Um exemplo de expressão lambda que define o operador de ponto fixo é:

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