Programação Funcional - Introdução a Cálculo Lambda

Cristiano Damiani Vasconcellos

Universidade do Estado de Santa Catarina

1. Definição

Cálculo lambda (λ-cálculo) é um modelo matemático capaz de ilustrar de forma simples alguns importantes conceitos presentes em linguagens de programação, como 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)
	| $\lambda x.$<expressão>	(abstração lambda ou função)

onde x pode ser uma variável (normalmente representada por ..., x, y, z, ...) ou uma constante. Exemplos de expressões lambda:

$\lambda x.x$	função identidade.
$\lambda f.\lambda g.\lambda x.f(gx)$	composição de funções.
$(\lambda x.x)5$	aplicação da função identidade.

É comum, para que as expressões fiquem mais concisas, deixar de usar o ponto para separar abstrações lambda quando estas encontram-se aninhadas. Existem algumas convenções sintáticas para eliminar a necessidade de uso de parêntese na maioria dos casos. As mais importante são que as aplicações ocorrem da esquerda para a direita e que o escopo de uma λ-abstração se estende o mais a direita possível, por exemplo $\lambda x.xy$ deve ser lido como $\lambda x.(xy)$, não como $(\lambda x.x)y$. Em uma expressão $\lambda x.M$, M é chamado do escopo de x.

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:

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

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

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

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

2. Substituição e Equivalência

A substituição de todas ocorrências uma variável x por uma expressão M é representada por $[M/x]$, por exemplo, 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):

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

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

2.1. Reduções

Escrevemos $M \to N$, 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 esta na **forma normal**.

Em alguns casos, a avaliação de uma expressão pode fornecer mais de um caminho possível de reduções. Uma importante propriedade de λ-cálculo, chamada de **confluência**, é que independente da seqüência de reduções aplicada o resultado (forma normal da expressão) é sempre o mesmo. É importante observar que podem ocorrer casos onde um determinado caminho pode ocasionar uma seqüência infinita de reduções, nunca atingindo a forma normal.

2.2. 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** (*eager evaluation ou applicative-order*). Uma outra alternativa de avaliação é substituir o argumento sem avalia-lo, nesse caso o argumento será reduzido apenas se necessário, essa ordem de avaliação e chamada de **avaliação preguiçosa** ou **ordem normal** (*normal-order evaluation ou lazy evaluation*). Por exemplo:

Avaliação por Valor:

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

Avaliação Preguiçosa:

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

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

A avaliação de uma expressão pode resultar em uma expressão na forma normal ou a computação pode não terminar, vamos representar a segunda situação através do símbolo $\perp$. Note que $\perp$ não é propriamente um valor e não pode ser manipulado. Uma função f é dita estrita se $f\ \perp = \perp$[1] e não estrita se $f\ \perp$, em alguma situação, fornecer outro valor. Por exemplo, a função identidade é claramente estrita, mas $\lambda x \lambda y.x$ não é, pois $(\lambda x \lambda y.y)\ \perp\ (\lambda x.x)$ é avaliado como: $\lambda x.x$. A existencia de funções não estritas torna significante a escolha da ordem de avaliação. Outro exemplo, a expressão $(\lambda x \lambda y.y)((\lambda z.zz)(\lambda z.zz))(\lambda w.w)$ termina se adortamos a ordem de avaliação preguiçosa e não termina se adortarmos a ordem de avaliação por valor.

> ***Teorema de Church e Rosser:*** *Se v é o resultado da avaliação de uma expressão M aplicando a ordem de avaliação preguiçosa, então qualquer que seja a ordem de avaliação aplicada, ou o resuldado da avaliação é v ou a avaliação falha (não termina). Se a avaliação de M não termina usando a ordem de avaliação preguiçosa a avaliação não termina usando qualquer ordem de avaliação.*

3. Currificação

Podemos representar uma função que recebe dois argumentos como $\lambda x.(\lambda y.M)$, onde M é uma expressão lambda, possívelmente envolvendo x e y. Aplicando um único argumento a 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.

4. Numerais de Church

Os números naturais podem ser expressos através 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. Em λ-cálculo calculo podemos 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:

[1] Uma função é estrita se necessita que o valor de todos seus argumentos sejam completamente calculados.

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

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

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

o resultado é α-equivalente a representação de 1. A função de adição entre dois números naturais é definida como:

$$ADD = \lambda x \lambda y \lambda w \lambda u.xw(ywu)$$

Como exemplo podemos aplicar a função de adição para os números naturais 2 e 3:

$(\lambda x \lambda y \lambda w \lambda u.xw(ywu))(\lambda s \lambda z.s(s(z)))(\lambda s \lambda z.s(s(s(z))))$
$\rightarrow (\lambda y \lambda w \lambda u.(\lambda s \lambda z.s(s(z)))w(ywu))(\lambda s \lambda z.s(s(s(z))))$
$\rightarrow (\lambda y \lambda w \lambda u.(\lambda z.w(w(z)))(ywu))(\lambda s \lambda z.s(s(s(z))))$
$\rightarrow (\lambda y \lambda w \lambda u.w(w(ywu)))(\lambda s \lambda z.s(s(s(z))))$
$\rightarrow \lambda w \lambda u.w(w((\lambda s \lambda z.s(s(s(z))))wu))$
$\rightarrow \lambda w \lambda u.w(w((\lambda z.w(w(w(z))))u))$
$\rightarrow \lambda w \lambda u.w(w(w(w(w(u)))))$

5. Booleanos

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

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

Dessa forma as operações lógicas são definidas como:

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

6. λ-Cálculo Enriquecido

Qualquer linguagem de programação deve fornecer meios de condicionar a avaliação de uma expressão ao resultado da avaliação de outra expressão, na grande maioria das linguagens de programação encontramos alguma variação da construção sintática:

`if` <*expressão1*> `then` <*expressão2*> `else` <*expressão3*>

Onde a *expressão2* é avaliada caso a *expressão1* um resulte em um valor *verdade*, caso contrário a *expressão3* é avaliada. Note que a definição do valor *verdade* é justamente uma função que recebe dois argumentos e tem como resultado o primeiro, e *falsidade* uma função que recebe dois argumentos e tem como resultado o segundo, portanto construções como a expressão de seleção poderiam ser definidas em λ-cálculo puro. Embora qualquer computação possa se expressão através 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.
$\lambda x \lambda y.$`if` $x > y$ `then` x `else` y	função que retorna o maior entre dois elementos

7. Operador de Ponto Fixo

A maioria das linguagens de programação modernas permitem a definição de funções recursivas, uma função recursiva possui uma ou mais referências a si mesma na sua declaração. Por exemplo, a função que calcula o fatorial de um inteiro positivo:

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

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

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

O **ponto fixo** da função G é tal que $G(f) = f$, o ponto fixo de uma função pode ser definido em λ-cálculo através do operador FIX que apresenta a seguinte identidade:

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

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

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

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

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

A expressão lambda que define o operador de ponto fixo é:

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

Referências

Mitchell, J. (1996). *Foundations for Programming Languages*. MIT Press.

Mitchell, J. (2003). *Concepts in Programming Languages*. Cambridge Press.

Rojas, R. A tutorial introduction to the lambda calculus. Technical report, FU Berlin.

Watt, D. A. (1991). *Programming Languages Syntaxe and Semantics*. Prentice Hall.