# Cálculo–$\lambda$ e sua Aplicação no Processamento de Sinais de Áudio

João Cândido Dovicchi
Laboratório de Música Eletroacústica
Núcleo Avançado de Computação Sônica e Multimídia
Universidade Federal de Uberlândia
Uberlândia - MG - Brasil
*Tech. Report*

19 de Fevereiro de 2003

**Resumo**

Este artigo apresenta uma introdução ao cálculo–$\lambda$ e suas aplicações em processamento digital de sinais de áudio. Uma de suas vantagens é a forma como o cálculo–$\lambda$ é utilizado na implementação de linguagens funcionais, que por sua vez tem aplicação direta nos algoritmos de processamento de áudio. Uma vez que o processamento digital de sinais de áudio concentra-se na análise e operações sobre uma lista de dados digitais, o uso das linguagens funcionais de programação facilita a criação de algoritmos com esta finalidade.

## 1 Introdução

Os algoritmos de processamento de sinais de áudio são, em geral, operações sobre vetores finitos de $2^n$ elementos, onde $n$ é um inteiro positivo. Em termos de algoritmo, tais operações podem ser consideradas como operações sobre listas finitas de elementos. Este fato justifica a utilização da programação funcional para a implementação destas operações.

A aplicação de transformadas para análise e síntese de sinais sonoros é extremamente dependente de operações lineares entre seus coeficientes e dados de um vetor que representa o sinal digital [3, 4, 5, 6]. Neste aspecto, um dos maiores problemas é como lidar com a computação de grandes vetores no domínio do tempo ou espaço e sua respectiva alocação de memória.

A característica principal da linguagem funcional, que pode ajudar na solução do problema, é o fato das linguagens funcionais usar a avaliação sob demanda (*lazy evaluation*) em uma lista finita ou infinita de dados. Além disso, a facilidade de implementar-se algoritmos paralelos é uma vantagem das linguagens funcionais sobre as procedurais.

A compreensão da base matemática destas linguagens — o cálculo–$\lambda$ — é fundamental para a compreensão destas implementações.

# 2 O Cálculo-$\lambda$

Uma dos mais importantes conceitos matemáticos para a programação funcional foi, com certeza, o cálculo-$\lambda$, desenvolvido por Church e Kleene [1]. A teoria desenvolvida por ele tinha como princípio demonstrar a computabilidade efetiva de expressões. Entretanto, o mais impressionante produto de suas deduções foi o fato de que, reduzindo-se um processo a um conjunto de regras matemáticas simples, é possível reduzir-se as chances de erro.

O cálculo-$\lambda$ influenciou muitas linguagens de programação funcional. Lisp foi a primeira e o *pure* Lisp é verdadeiramente funcional. Outras linguagens atuais — Haskell, Miranda, ML e clean — foram desenvolvidas a partir do cálculo-$\lambda$, baseadas na definição formal e na semântica denotativa que ele permite.

Em 1940, Haskell B. Curry [2] formulou a teoria da lógica combinatória como uma teoria de funções livres de variáveis. Apesar de fundamentada na teoria de funções de Schönfinkel [8] e no cálculo-$\lambda$ de Church, sua teoria foi fundamental para o desenvolvimento das linguagens funcionais.

A sintaxe do cálculo-$\lambda$ é bastante simples. Ela está fundamentada em identificadores de variáveis, constantes, abstrações de funções, aplicação de funções e subexpressões.

Para compreender a teoria do cálculo-$\lambda$, é necessário que se observe com maiores detalhes as definições das fórmulas matemáticas. Para isso vamos começar com o conceito de identificador delimitado.

Em uma equação, por exemplo,

$$f(x) = x^2 \tag{1}$$

podemos dizer que $x$ é uma variável e a expressão $f(x)$ assumirá valores que dependem do valor de $x$. Ora, esta relação é ambígua, pois se quisermos determinar se $f(x) \leq 25$ a variável $x$ somente poderá assumir valores de 0 a 5, pois para $x > 5$, o resultado será falso. Então temos que explicitar que $x^2 \leq 25 \iff \exists x | 0 \leq x \leq 5$, ou seja $x^2 \leq 25$ se existir $x$ tal que $x$ é maior ou igual a 0 e menor ou igual a 5. Entretanto, se quisermos determinar se $f(x) < x^3$, o resultado será sempre verdadeiro, independentemente do valor que $x$ venha a ter.

Vamos, então, observar estas entidades matemáticas com maior detalhe. Na equação,

$$\sum_{x=1}^{n} x^2 \tag{2}$$

$x$ assumirá valores de $[1 \cdots n]$ e seus quadrados serão somados, ou seja, $[1 + 4 + 9 + \cdots + n^2]$. Neste caso, $x$ pode ser chamado de uma variável dependente ou um identificador que está delimitado (de 1 a $n$). Note que um identificador delimitado (*bound identifier*) faz com que qualquer das expressões das equações 3, sejam equivalentes à equação 2.

$$\sum_{y=1}^{n} y^2, \quad \sum_{k=1}^{n} k^2, \quad \sum_{a=1}^{n} a^2, \cdots \tag{3}$$

Da mesma forma, na teoria dos conjuntos, podemos escrever $x | x \geq 0 = y | y \geq 0$ ou, ainda, $\forall x[x + 1 > x]$ é equivalente a $\forall y[y + 1 > y]$.

Quando temos uma expressão do tipo:

$$f(x) = x^2 + 4 \tag{4}$$

dizemos que o parâmetro formal $x$ é o identificador delimitado, ou seja, a função retornará algum valor para para qualquer que seja o valor de $x$ de forma indefinida. Pode-se perceber facilmente que para $-\infty \leq x \leq \infty$ a função retornará valores entre 4 e $\infty$.

Podemos definir os termos do cálculo-$\lambda$ ($\lambda$-termo[1].) como termos construídos a partir de recursividades sobre variáveis $x, y, z, \ldots$ que podem assumir as formas descritas na figura 1

Em $(\lambda x.M)$, $x$ é denominada variável dependente (*bounded variable*) e $M$ é o termo que sofre a abstração. Cada ocorrência de $x$ em $M$ está vinculada à abstração. Assim, podemos escrever que:

[1] Em programação usa-se `\x` para representar $\lambda$x

| | |
|---|---|
| $x$ | `variável` |
| $\lambda.M$ | `abstração onde M é um termo` |
| $(M, N)$ | `aplicação onde M e N são termos` |

Figura 1: Formas dos termos no cálculo-$\lambda$.

| | | |
|---|---|---|
| $(\lambda x.N)M$ | $\equiv$ | `let x = M in N` |
| $(\lambda f.N)(\lambda x_1 \ldots x_k.M)$ | $\equiv$ | `let f x1 ... xk = M in N` |

Assim, podemos representar a equação 4, em termos do cálculo-$\lambda$, como:

$$\lambda x(x^2 + 4) \tag{5}$$

e, se $f \equiv \lambda x(x^2 + 4)$, podemos representar $f(2)$ como:

$$\lambda x(x^2 + 4)(2) \tag{6}$$

## 2.1 Cópia e reescrita

O cálculo é feito substituindo-se todas as ocorrências de $x$ por 2, ou seja, $(2^2 + 4) \Rightarrow 8$. Este tipo de operação é denominado "regra de cópia" (*copy rule*), uma vez que para $f(2)$ copiamos o valor 2 em todas as ocorrências de $x$. No cálculo-$\lambda$ existem apenas três símbolos: $\lambda, (, )$ e identificadores. Estes são regidos por quatro regras básicas:

1. Se $x$ é um identificador, então $x$ é uma expressão do cálculo-$\lambda$;

2. Se $x$ é um identificador e E é uma expressão do cálculo-$\lambda$, então $\lambda x E$ é uma expressão do cálculo-$\lambda$, chamada "abstração". Assim, $x$ é o identificador que delimita a abstração e E o corpo da abstração.

3. Se $F$ e $E$ são expressões do cálculo-$\lambda$, então $FE$ é uma expressão do cálculo-$\lambda$, chamada "aplicação". Assim, $F$ é o operador da aplicação e $E$ o seu operando.

4. Uma expressão $\lambda$ é produzida, sempre, pela aplicação múltipla e finita das regras 1–3.

Como estas regras operam por recursividade, podemos dizer que, uma vez que esta abstração é definida em um determinado domínio, ela pode ser aplicada em qualquer entidade. Por exemplo, digamos que queiramos aplicá-la ao domínio de $z^2$. Então, podemos escrevê-la como:

$$\lambda x(2x+3)(z^2) \tag{7}$$

A equação 7 pode ser, então, re-escrita por $2z^2+3$. Note, por exemplo que se quisermos expressar a função de adição $f(a,b)=a+b$, podemos escrever: $\lambda ab(a+b)$ que, por sua vez, pode ser re-escrita por $\lambda a(\lambda b(a+b))$, que é denominado "currying"[2].

A esta altura voce deve estar pensando, mas que vantagens há nisto? Não seria melhor usar a velha e tradicional notação? Vejamos! Uma vez que podemos representar $f(x)$ simplesmente por $fx$ e, sendo a aplicação da abstração sempre associativa à esquerda, podemos evitar as confusões de parênteses no caso de $f(x)(y)$, $(f(x))y$ e representar apenas por $fxy$. Outra vantagem é que podemos escrever a identidade como $\lambda a(a)$ e uma identidade que aplica uma combinação em si mesma como $\lambda a(aa)$. Veja, por exemplo, como uma combinação que inverte a ordem de dois combinadores pode, facilmente, ser expressa por $\lambda ab(ba)$.

## 2.2 Redução

As expressões do cálculo-$\lambda$ são computadas por "redução". Por exemplo, digamos que seja preciso avaliar a expressão:

$$\lambda y(y^2)(\lambda x(x^3+2)(2)) \tag{8}$$

A equação 8 pode ser calculada por substituições e reduções sucessivas, da seguinte forma:

$$\begin{array}{rcl}
\lambda y(y^2)(\lambda x(x^3+2)(2)) & \Rightarrow & \lambda y(y^2)(2^3+2) \\
\lambda y(y^2)(2^3+2) & \Rightarrow & \lambda y(y^2)(10) \\
\lambda y(y^2)(10) & \Rightarrow & (10^2) \\
(10^2) & \Rightarrow & 100
\end{array}$$

A semântica do cálculo-$\lambda$ estabelece duas regras para a redução, na computação de expressões:

[2]Essa transformação é possível porque $\lambda ab(a)=\lambda a(\lambda b(a))$. O nome "currying" é usado por ter, essa expansão, sido definida por Haskell B. Curry.

1. Regra de renomeação: Uma expressão pode ser reduzida a outra pela renomeação de um identificador limitado, por qualquer outro identificador que não esteja contido na expressão.

2. Regra de substituição: Uma subexpressão da forma $(\lambda x E)A$ pode ser reduzida pela substituição de uma cópia de $E$ na qual toda ocorrência livre de $x$ é substituída por $A$, desde que isto não resulte em que qualquer identificador livre de $A$ torne-se um delimitador.

Estas regras são conhecidas por redução-$\alpha$ e redução-$\beta$, respectivamente. Para compreendê-las melhor, vamos detalhar um pouco mais estas regras.

A primeira pode ser exemplificada da seguinte maneira: seja uma expressão $\lambda xM$ que pode ser renomeada pela expressão $\lambda yN$, onde $N$ foi obtida pela renomeação do identificador $x$ em $M$ pelo identificador $y$. Isto só é possível se $y$ não ocorrer em $M$. Por exemplo:

$$\lambda x(2x+z) \Rightarrow \lambda y(2y+z) \tag{9}$$

Uma vez que $y$ não ocorre à direita da equação 9, ele pode ser usado para renomear $x$. O mesmo não seria possível no caso de renomear $x$ por $z$, o que resultaria em erro[3].

Na prática, imagine uma função:

$$f = \lambda x(\lambda z(z+x)(4)) \Rightarrow f = \lambda x(4+x)$$

Imagine, agora, a aplicação errada da redução-$\alpha$ renomeando $x$ para $z$:

$$f = \lambda x(\lambda z(z+x)(4)) \Rightarrow f' = \lambda z(\lambda z(z+z)(4)) = \lambda z(8)$$

Observe que $f'$ não é a mesma função $f$, pois $f(1) = 5$ e $f'(1) = 8$.

No caso da regra de redução-$\beta$ vamos considerar a expressão $\lambda x(2x+1)(3)$. Neste caso, a regra de substituição tem um operador que é uma abstração e pode ser reduzido pela substituição de 3 em toda a ocorrência de $x$ em $2x+1$, o que resulta em $2 \times 3 + 1 = 7$. No caso de

$$\lambda x(\lambda y(xy))n$$

a aplicação é uma abstração $\lambda x(\lambda y(xy))$ e usando-se a redução-$\beta$ em todas as ocorrências livres de $x$ teremos:

[3]Neste caso a redução resultaria em $\lambda z(2z)$.

$$\lambda y(ny)$$

Vejamos, então a restrição da regra da redução-$\beta$. Considere a aplicação:

$$\lambda y(\lambda x(\lambda z(z+x)(3))(y))(1) \tag{10}$$

Neste caso, podemos aplicar a redução:

$$\begin{aligned}
\lambda y(\lambda x(\lambda z(z+x)(3))(y))(1) &\Rightarrow \lambda y(\lambda z(z+y)(3))(1)\\
\lambda y(\lambda z(z+y)(3))(1) &\Rightarrow \lambda z(z+1)(3)\\
\lambda z(z+1)(3) &\Rightarrow 3+1\\
3+1 &\Rightarrow 4
\end{aligned}$$

Note que no primeiro passo da redução-$\beta$ da equação 10 o identificador $x$ foi substituído por $y$, então $y$ foi substituído por 1 e, finalmente, $z$ por 3. Vamos, entretanto, observar o que acontece se no primeiro passo substituímos $z$ por $y$:

$$\begin{aligned}
\lambda y(\lambda x(\lambda z(z+x)(3))(y))(1) &\Rightarrow \lambda y(\lambda x(\lambda y(y+x)(3))(y))(1)\\
\lambda y(\lambda x(\lambda y(y+x)(3))(y))(1) &\Rightarrow \lambda y(\lambda y(y+y)(3))(1)\\
\lambda y(\lambda y(y+y)(3))(1) &\Rightarrow \lambda y(6)(1)\\
\lambda y(6)(1) &\Rightarrow 6
\end{aligned}$$

O que aconteceu? Acontece que, quando $z$ foi substituído por $y$, no primeiro passo, houve uma colisão de identificadores, uma vez que $y$ é um identificador livre em $y$ mas não em $\lambda y(y+x)(3)$. A partir daí, houve uma mudança no significado da expressão, resultando em um valor diferente do esperado. Note a importância desta regra de redução para se evitar colisões entre os identificadores.

## 2.3 Forma normal

Chamamos de “forma normal” a máxima redução possível de uma expressão, ou seja, a regra de substituição não pode mais ser aplicada. A computabilidade de uma expressão depende de sua forma normal ser computável, isto é, quando uma expressão atinge sua forma normal como uma resposta. A tabela 1 mostra alguns exemplos de expressões em sua forma não-normal e normal.

| **Não-normal** | **Normal** |
|---|---|
| $\lambda x(x)(y)$ | $y$ |
| $\lambda y(fy)(a)$ | $fa$ |
| $\lambda x(\lambda y(xy))(f)$ | $\lambda y(fy)$ |
| $\lambda x(xx)(y)$ | $yy$ |

Tabela 1: Exemplos de forma normal, cf. MacLennan [7].

# 3 Aplicações

Embora possa haver aplicações diretas do cálculo-$\lambda$ no cálculo de transformadas, sua aplicação está mais ligada à compreensão das bases teóricas das linguagens funcionais.

O uso de linguagens funcionais para implementar algoritmos de processamento digital de sinais, principalmente sinais de áudio, é uma escolha quase natural. Uma das grandes vantagens das linguagens funcionais é a maneira pela qual elas lidam com listas de dados. Estas funções se aplicam às operações dos coeficientes de análise das transformadas aos vetores que representam os dados digitalizados de um sinal de áudio, considerando-os como listas finitas.

As combinações lineares de coeficientes e dados dos sinais representam um grande esforço computacional que pode ser minimizado pela construção de algoritmos discretos e, ainda, o uso de paradigmas de computação paralela. Tais implementações são facilmente programadas em linguagens funcionais, possibilitando o tratamento de grande quantidade de dados com menor esforço e em menor tempo de processamento.

# Referências


[1] Church, A. "An Unsolvable Problem of Elementary Number Theory", *Am. J. Math.*, 58: 345–363, 1936.

[2] Curry, H. B.; Feys, R.; e Craig, W. "Combinatory Logic", *in:* **Studies in Logic and the Foundations of Mathematics**, Vol. I, Amsterdan: North-Holland, 1958.

[3] Dovicchi, J. C. L.; Carrijo G.; e Lima, L. "Short Time Fourier Transforms And Wavelets: A Combined Technique For Musical Signal Analysis", *In*:

**Conference on Mathematical and Computational Methods in Music**, 1999, Viena, Diderot'99. Viena - Austria, 1999.

[4] Dovicchi, J. C. L., Carrijo, G.; e Lima, L. "Music Based Coefficients for Wavelets Transform: a Contribution to Musical Signal Analysis" *In*: **108th AES Convention**, 2000, Paris. 108th. Convention of the Audio Engineering Society, 2000.

[5] Dovicchi, J. C. L. "Introdução à Teoria das Wavelets", *Tech. Rep.*, Lab. Eletroacústica, NACSM/UFU, 2003. `http://www.lab.demac.ufu.br/lab/papers-jcd/wav-intro.ps`.

[6] Dovicchi, J. C. L.; Parreira, W. D. "Combinações Lineares de Vibrações Harmônicas Simples e Complexas", *Tech. Rep.*, Lab. Eletroacústica, NACSM/UFU, 2003. `http://www.lab.demac.ufu.br//lab/papers-jcd/acoust.ps`

[7] MacLennan, B. J. **Functional Programming — Practice and Theory**, Reading, Massachusetts: Addison-Wesley Pub. Co., 1990.

[8] Schönfinkel, M. "Über die Bausteine der mathematischen Logik", *Mathematische Annalen*, 92: 305–316, 1924.