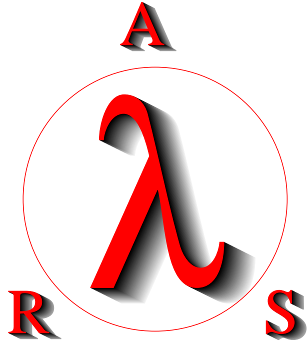
The Lambda Calculus

A Brief Introduction

Georg P. Loczewski



The Lambda Calculus

A Brief Introduction

stmv

S.Toeche-Mittler Verlag

founded in 1789

Darmstadt, Germany

Loczewski, Georg P.
The Lambda Calculus: A Brief Introduction
1st. Edition 2005 –Darmstadt
STMV – S. Toeche-Mittler Verlag

S. Toeche-Mittler Versandbuchhandlung GmbH
www.net-library.de
www.aplusplus.net
orders@net-library.de

© 2005 by S. Toeche-Mittler Verlag, 64295 Darmstadt. All rights reserved.

This article is an excerpt from the book *A++ The Smallest Programming Language in the World, An Educational Programming Language* published by the S.Toeche-Mittler Verlag in Darmstadt in October 2004 under the **ISBN 3-87820-116-8**.

Permission is granted to copy, distribute, and/or modify this article under the terms of the GNU Free Documentation License published by the Free Software Foundation; with the Front-Cover Texts being "The Lambda Calculus: A Brief Introduction, by Georg P. Loczewski published by the S.Toeche-Mittler Verlag" and with no Back-Cover Texts. The full text of the license is available at the following WWW-address: `http://www.gnu.org/copyleft/fdl.html`.

The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with the use of these programs.

Contents

1 Introduction **1**
1.1 Origin . . . 1
1.2 Definition . . . 1
1.3 Literature . . . 1

2 Syntax of Lambda Expressions **2**

3 Basic Rules for Lambda Conversions **3**
3.1 Notation used in Conversion Rules . . . 3
3.1.1 Notation used to specify conversion of lambda expressions . . . 3
3.1.2 Notation used to specifiy substitution . . . 3
3.2 Alpha Conversion . . . 4
3.2.1 Bound and Free Variables . . . 4
3.2.2 Rules of Alpha Conversion . . . 4
3.3 Beta Conversion . . . 4
3.3.1 Rule of β-Conversion . . . 5
3.3.2 β-Reduction . . . 5
3.3.3 β-Abstraction . . . 5
3.4 Eta Conversion . . . 6
3.4.1 η-Reduction . . . 6
3.4.2 η-Abstraction . . . 6
3.5 Rules of Associativity . . . 7
3.5.1 Rule of Associativity for Abstraction . . . 7
3.5.2 Rule of Associativity for Application . . . 7
3.5.3 Example for both rules: . . . 7
3.6 Y-Combinator . . . 7
3.6.1 Definition . . . 7
3.6.2 Basic Usage of Y-Combinator . . . 8
3.6.3 Using the Y-Combinator to implement recursion . . . 9

Bibliography **11**

Chapter 1

Introduction

1.1 Origin

The **Lambda Calculus** has been created by the American logician **Alonzo Church** in the 1930's and is documented in his works published in 1941 under the title *'The Calculi of Lambda Conversion'*.

Alonzo Church wanted to formulate a mathematical logical system and had no intent to create a programming language. The intrinsic relationship of his system to programming was discovered much later in a time in which programming of computers became an issue.

1.2 Definition

> DEFINITION 1 (LAMBDA CALCULUS)
> The Lambda Calculus defines the laws for the formulation and conversion of lambda expressions.

1.3 Literature

As a *mathematical logical system* the Lambda Calculus is covered in detail in [Bar81] and less comprehensively but in a more readable form in [Sto81]. A clear account of the historical origins and basic properties of the lambda calculus is presented by Curry and Fey in their book [CF58]. This view is taken from [Jon87] page 23.

From the programmer's point of view the Lambda Calculus is adressed in [Jon87], [Kam90], [Cha96].

Chapter 2

Syntax of Lambda Expressions

The syntax of lambda expressions is defined as follows:

DEFINITION 2 (SYNTAX OF A LAMBDA EXPRESSION)
t is a lambda expression, if

- $t = x$ where $x \in Var$,
  or
- $t = \lambda x.M$ where $x \in Var$ and M is a lambda expression,
  or
- $t = (MN)$ where M and N are lambda expressions.

The Lambda Calculus therefore includes three diffenrent types of lambda expressions:

- **variables** (referencing lambda expressions)
- **lambda abstractions** (defining functions)
- **applications** (invoking functions)

Remark:
The parentheses in the syntax of an application are not mandatory. This results from the law of associativity for applications introduced below.

Chapter 3

Basic Rules for Lambda Conversions

3.1 Notation used in Conversion Rules

The notation used to specify a conversion of a lambda expression and associated with it substitutions to be performed varies from textbook to textbook. In 'Programmierung pur' we adopted the notation used by Chazarain in [Cha96]. For greater clarity and flexibility we prefer to use the following notation here:

3.1.1 Notation used to specify conversion of lambda expressions

NOTATION 1 (CONVERSION OPERATOR)

$M \underset{\alpha}{\rightarrow} N$ rule applies to conversion from left to right

$M \underset{\alpha}{\leftarrow} N$ rule applies to conversion from right to left

$M \underset{\alpha}{\leftrightarrow} N$ rule applies to conversion in both directions

where the Greek letter below the arrow specifies the type of conversion.

3.1.2 Notation used to specifiy substitution

NOTATION 2 (SUBSTITUTION OPERATOR)

$M[x{\rightarrow}N]$ x in M is substituted by N

$M[x{\leftarrow}N]$ N in M is substituted by x

$M[x{\leftrightarrow}N]$ x in M is substituted by N
or N in M is substituted by x
depending on the direction of conversion

3.2 Alpha Conversion

3.2.1 Bound and Free Variables

DEFINITION 3 (BOUND AND FREE VARIABLES)
In the following lambda expression:

$$\lambda x.xy$$

"y" is a free variable whereas "x" is a bound variable. Free variables take their values from lambda expressions on higher levels.

3.2.2 Rules of Alpha Conversion

An Alpha Conversion of a lambda expression is defined as follows:

RULES 1-3 (ALPHA CONVERSION)

$$\lambda x.M \underset{\alpha}{\rightarrow} \lambda x_0.M[x \rightarrow x_0] \qquad 1$$

$$\lambda x.M \underset{\alpha}{\leftarrow} \lambda x_0.M[x \leftarrow x_0] \qquad 2$$

$$\lambda x.M \underset{\alpha}{\leftrightarrow} \lambda x_0.M[x \leftrightarrow x_0] \qquad 3$$

where x_0 is not allowed to be a free variable in M.
The process of alpha conversion may not alter the value of the expression. The expression to be converted (M) and the converted result (N) are said to be equal *modulo alpha*:

$$M =_\alpha N.$$

The third rule for alpha conversion is a combination of the first two rules.

Alpha conversion may be required, if substitutions are performed in lambda expressions.

3.3 Beta Conversion

β-Conversion primarily consists of the process of substituting a bound variable in the body of a lambda abstraction by the argument passed to the function whenever it is applied. This process is called *β-reduction*.

The inverse process to convert a β-reduced lambda expression back to the reducible expression is another aspect of β-conversion and is called *β-abstraction*.

3.3.1 Rule of β-Conversion

RULE 4 (BETA CONVERSION)
The following transformation is called β-conversion:

$$((\lambda x.M)N) \underset{\beta}{\leftrightarrow} M[x \leftrightarrow N]$$

3.3.2 β-Reduction

Reducible Expression 'redex' β-reduction can be applied only to reducible expressions. A reducible expression called 'redex' for short is defined as follows:

DEFINITION 4 (REDEX)
A redex is a reducible expression and is represented as:

$$((\lambda x.M)N)$$

Rule of β-Reduction

RULE 5 (BETA REDUCTION)
The following transformation is called β-reduction:

$$((\lambda x.M)N) \underset{\beta}{\rightarrow} M[x \rightarrow N]$$

3.3.3 β-Abstraction

Rule of β-Abstraction

RULE 6 (BETA ABSTRACTION)
The following transformation is called β-abstraction:

$$((\lambda x.M)N) \underset{\beta}{\leftarrow} M[x \leftarrow N]$$

Examples:

- $((\lambda x.x\ x)(\lambda y.y)) \rightarrow_\beta ((\lambda y.y)(\lambda y.y)) \rightarrow_\beta (\lambda y.y)$
- $((\lambda x.(\lambda y.x\ y))y) \rightarrow_\beta (\lambda y_0.y\ y_0)$

Remark: The second example demonstrates the necessity of alpha conversion. The lambda bound variable y had to be renamed y_0, to prevent capturing of the free y (resulting from the substitution of x by y in the body of the first lambda absstraction) by the second lambda.

3.4 Eta Conversion

Like β-conversion η-conversion can be performed from left to right and from right to left and is therefore subdivided in

- η-reduction and
- η-abstraction

> RULE 7 (ETA CONVERSION)
> The following transformation of a lambda expression is called η-conversion.
>
> $$(\lambda x.Mx) \underset{\eta}{\leftrightarrow} M \ ,$$
>
> where x may not be a free variable in M.

3.4.1 η-Reduction

η-reduction is useful to eliminate redundant lambda abstractions. The following rule can be interpreted like this:

If the sole purpose of a lambda abstraction is to pass its argument to another function, then the lambda abstraction is redundant and can be stripped via η-reduction.

In an environment where *'eager evaluation'* is used like in Scheme such redundant lambda abstractions are used as a wrapper around a lambda expression to prevent immediate evaluation.

> RULE 8 (ETA REDUCTION)
> The following transformation of a lambda expression is called η-reduction.
>
> $$(\lambda x.Mx) \underset{\eta}{\rightarrow} M \ ,$$
>
> where x may not be a free variable in M.

3.4.2 η-Abstraction

η-abstraction on the contrary is useful in 'eager' languages to create a wrapper around a lambda-expression. In 'lazy' languages like Lambda Calculus, A++, SML, Haskell, Miranda etc. η-conversion, abstraction and reduction alike, are mainly used within compilers. (See [Jon87] page 22.)

> RULE 9 (ETA ABSTRACTION)
> The following transformation of a lambda expression is called η-abstraction.
>
> $$(\lambda x.Mx) \underset{\eta}{\leftarrow} M \ ,$$
>
> where x may not be a free variable in M.

3.5 Rules of Associativity

3.5.1 Rule of Associativity for Abstraction

RULE 10 (RULE OF ASSOCIATIVITY FOR ABSTRACTION) Abstraction is associative from left to the right.

Example: The expression

$$\lambda x.\lambda y.\lambda z.M$$

can be rewritten as:

$$\lambda xyz.M$$

3.5.2 Rule of Associativity for Application

RULE 11 (RULE OF ASSOCIATIVITY FOR APPLICATION) Application (Synthesis) is associative from right to left.

Example: The expression:

$$((MN)P)$$

can be rewritten as:

$$MNP.$$

3.5.3 Example for both rules:

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

is equivalent to

$$\lambda xy.xyz.$$

3.6 Y-Combinator

3.6.1 Definition

The “Y-combinator”, which is sometimes also called *fixpoint combinator*, was discovered by *H. Curry*. With its help it is possible to *handle recursive functions* in the Lambda Calculus.

In many programming languages it is possible to simply refer to the name of a function within the function itself. This is called *implicit recursion*, which is not possible in the Lambda Calculus because all lambda abstractions are by definition ‘anonymous functions’.

DEFINITION 5 (Y-COMBINATOR)
The Y-Combinator is defined as follows:

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

3.6.2 Basic Usage of Y-Combinator

Fixpoint of a Function If it can be shown that $(M\ N) =_\beta N$ is true, then N is called either a *'fixed point'* or a *'fixpoint'* of M.

According to H. Curry, there exists a function, that generates such a *fixed point* of M. This function is the so-called Y-combinator introduced above.

DEFINITION 6 (FIXPOINT OF A FUNCTION)

$$(MN) \underset{\beta}{\rightarrow} N$$

According to the above definition a fixpoint of a function can be seen as a lambda expression that, if passed as argument to this function is returned by the function.

Generation of fixpoint using Y-combinator By definition the Y-combinator has the function to generate a fixpoint of M:

RULE 12 (FIXPOINT GENERATION USING Y-COMBINATOR)

$$(M(Y\ M)) \underset{\beta}{\rightarrow} (Y\ M)$$

Fixpoint expansion by $_\beta$-reduction It looks like magic that expansion can be obtained by reduction as well as by abstraction. The first of these alternatives is shown below:

RULE 13 (FIXPOINT EXPANSION)

$$(Y\ M) \underset{\beta}{\rightarrow} (M(Y\ M))$$

Verification of the expansion formula:

PROOF 1 (FIXPOINT EXPANSION)

$$\begin{aligned}(Y\ M) &\rightarrow_\beta ((\lambda x.(M(x\ x)))(\lambda x.(M(x\ x))))\\ &\rightarrow_\beta (M((\lambda x.(M(x\ x)))(\lambda x.(M(x\ x)))))\\ &\rightarrow_\beta (M(Y\ M)).\end{aligned}$$

3.6.3 Using the Y-Combinator to implement recursion

Introduction The following classical example for recursive programming, the function to calculate the factorial of a natural number, is used to test the Y-combinator.

In order to simplify the procedure we use the functions $IF, =, *, -$ as predefined 'lambda abstractions' and assume as well the availability of all natural numbers as values. To make these definitions in the Lambda Calculus would not be difficult, but it would certainly distract from our present topic.

Steps to Eliminate Implicit Recursion

1. recursive function to calculate the factorial including illegal implicit recursion:
   $FAC = \lambda n.(IF(= n\ 0)1(*n(FAC(-n\ 1))))$
2. locating illegal implicit recursion:
   $FAC = \lambda n.(...FAC...)$
3. eliminating illegal implicit recursion:
   $FAC = (\lambda fac.\lambda n.(\ldots fac \ldots)FAC)$
4. simplified representation:
   $FAC = (M\ FAC)$
   where:
   $M = \lambda fac.\lambda n.(...fac...)$
   The recursion is eliminated by enclosing the recursive function in a lambda abstraction, that receives the function to be invoked recursively as an argument.
5. Using definition 12 and the result in step 4 FAC is identified as a *fixed point* of M.
   Therefore we can write:
   $FAC = (Y\ M)$
6. The function to calculate the factorial can now be rewritten as:
   $FAC = (Y\ \lambda fac.\lambda n.(...fac...))$
7. To compute the factorial of n the following expression has to be evaluated:
   $((Y\ \lambda fac.\lambda n.(...fac...)\ )n)$

Example of Step by Step Evaluation

- predefined variables:
  - $FAC = (YM)$
  - $M = \lambda fac.\lambda n.(IF(= n\ 0)1(*n(fac(-n\ 1))))$
- expression to be evaluated:
  $(FAC\ 1)$

- expression rewritten replacing FAC, a fixed point of M, by (YM):

  $((YM)1)$

- expression rewritten using rule 12:

  $((M(YM))1)$

- expression rewritten substituting the 1st occurrence of M::

  $((\lambda fac.\lambda n.(IF(= n\ 0)1(*n(fac(-n\ 1))))(YM))1)$

- performing $_\beta$-reduction by substituting variable fac in the body of the lambda abstraction by the argument $(Y\ M)$:

  $(\lambda n.(IF(= n\ 0)1(*n((YM)(-n\ 1))))1)$

- performing $_\beta$-reduction by substituting variable n in the body of the lambda abstraction by the argument 1:

  $(IF(= 1\ 0)1(*1((YM)(-1\ 1))))$

- evaluating the 'IF'-expression by selecting the 'no'-branch and evaluating the subtraction:

  $(*1((YM)0))$

- expression rewritten substituting (YM) by $(M(YM))$ using rule 12:

  $(*1((M(YM))0))$

- expression rewritten substituting the first occurrence of the variable M by its value:

  $(*1((\lambda fac.\lambda n.(IF(= n\ 0)1(*n(fac(-n\ 1))))(YM))0))$

- performing $_\beta$-reduction substituting 'fac' in the body of the lambda abstraction by the argument (YM):

  $(*1(\lambda n.(IF(= n\ 0)1(*n((YM)(-n\ 1)))))0)$

- performing $_\beta$-reduction substituting n in the body of the lambda abstraction by the argument 0:

  $(*1((IF(= 0\ 0)1(*0((YM)(-0\ 1))))))$

- evaluating 'IF'-expression returning the 'yes'-branch:

  $(*1\ 1)$

- evaluating the multiplication:

  1

Bibliography

[Bar81] H. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*. North-Holland, Amsterdam, 1981.

[CF58] Howard B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

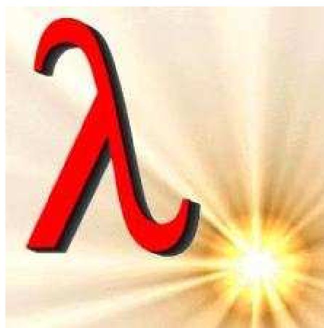
[Cha96] Jacques Chazarain. *Programmer avec Scheme – De la pratique à la théorie*. International Thomson Publishing France, Paris, 1996. ISBN 2 84180 130 4.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, Hertfordshire,HP2 7EZ, 1987. ISBN 0 13 453325 9.

[Kam90] Samuel N. Kamin. *Programming Languages – An Interpreter-Based Approach*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1990. ISBN 0 201 06824 9.

[Sto81] J.E. Stoy. *Denotational Semantics*. MIT Press, Cambridge, Massachusetts, 1981.



The Lambda Calculus
A Brief Introduction
http://www.aplusplus.net — http://www.net-library.de