A Tutorial Introduction to the Lambda Calculus

Raúl Rojas*

FU Berlin, WS-97/98

Abstract

This paper is a short and painless introduction to the λ calculus. Originally developed in order to study some mathematical properties of effectively computable functions, this formalism has provided a strong theoretical foundation for the family of functional programming languages. We show how to perform some arithmetical computations using the λ calculus and how to define recursive functions, even though functions in λ calculus are not given names and thus cannot refer explicitly to themselves.

1 Definition

The λ calculus can be called the *smallest universal programming language of the world*. The λ calculus consists of a single transformation rule (variable substitution) and a single function definition scheme. It was introduced in the 1930s by Alonzo Church as a way of formalizing the concept of effective computability. The λ calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to Turing machines. However, the λ calculus emphasizes the use of transformation rules and does not care about the actual machine implementing them. It is an approach more related to software than to hardware.

The central concept in λ calculus is the "expression". A "name", also called a "variable", is an identifier which, for our purposes, can be any of the letters $a, b, c, \ldots$ An expression is defined recursively as follows:

<expression>	:=	<name> | <function> | <application>
<function>	:=	λ <name>.<expression>
<application>	:=	<expression><expression>

An expression can be surrounded with parenthesis for clarity, that is, if E is an expression, (E) is the same expression. The only keywords used in the language are λ and the dot. In order to avoid cluttering expressions with parenthesis, we adopt the convention that function application associates from the left, that is, the expression

$$E_1 E_2 E_3 \ldots E_n$$

*Send corrections or suggestions to rojas@inf.fu-berlin.de

is evaluated applying the expressions as follows:

$$(\ldots((E_1E_2)E_3)\ldots E_n)$$

As can be seen from the definition of λ expressions given above, a single identifier is a λ expression. An example of a function is the following:

$$\lambda x.x$$

This expression defines the identity function. The name after the λ is the identifier of the argument of this function. The expression after the point (in this case a single x) is called the "body" of the definition.

Functions can be applied to expressions. An example of an application is

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

This is the identity function applied to y. Parenthesis are used for clarity in order to avoid ambiguity. Function applications are evaluated by substituting the value of the argument x (in this case y) in the body of the function definition, i.e.

$$(\lambda x.x)y = [y/x]x = y$$

In this transformation the notation $[y/x]$ is used to indicate that all occurrences of x are substituted by y in the expression to the right.

The names of the arguments in function definitions do not carry any meaning by themselves. They are just "place holders", that is, they are used to indicate how to rearrange the arguments of the function when it is evaluated. Therefore

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

and so forth. We use the symbol "$\equiv$" to indicate that when $A \equiv B$, A is just a synonym of B.

1.1 Free and bound variables

In λ calculus all names are local to definitions. In the function $\lambda x.x$ we say that x is "bound" since its occurrence in the body of the definition is preceded by λx. A name not preceded by a λ is called a "free variable". In the expression

$$(\lambda x.xy)$$

the variable x is bound and y is free. In the expression

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

the x in the body of the first expression from the left is bound to the first λ. The y in the body of the second expression is bound to the second λ and the x is free. It is very important to notice that the x in the second expression is totally independent of the x in the first expression.

Formally we say that a variable <name> is free in an expression if one of the following three cases holds:

- <name> is free in <name>.
- <name> is free in λ <name$_1$ > . <exp> if the identifier <name>$\neq$<name$_1$ > and <name> is free in <exp>.
- <name> is free in E_1E_2 if <name> is free in E_1 or if it is free in E_2.

A variable <name> is bound if one of two cases holds:

- <name> is bound in λ <name$_1$ > . <exp> if the identifier <name>=<name$_1$ > or if <name> is bound in <exp>.
- <name> is bound in E_1E_2 if <name> is bound in E_1 or if it is bound in E_2.

It should be emphasized that the same identifier can occur free and bound in the same expression. In the expression

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

the first y is free in the parenthesized subexpression to the left. It is bound in the subexpression to the right. It occurs therefore free as well as bound in the whole expression.

1.2 Substitutions

The more confusing part of standard λ calculus, when first approaching it, is the fact that we do not give names to functions. Any time we want to apply a function, we write the whole function definition and then procede to evaluate it. To simplify the notation, however, we will use capital letters, digits and other symbols as synonyms for some function definitions. The identity function, for example, can be denoted with I which is a synonym for $(\lambda x.x)$.

The identity function applied to itself is the application

$$\mathsf{II} \equiv (\lambda x.x)(\lambda x.x)$$

In this expression the first x in the body of the first expression in parenthesis is independent of the x in the body of the second expression. We can in fact rewrite the above expression as

$$\mathsf{II} \equiv (\lambda x.x)(\lambda z.z)$$

The identity function applied to itself

$$\mathsf{II} \equiv (\lambda x.x)(\lambda z.z)$$

yields therefore

$$[\lambda z.z/x]x = \lambda z.z \equiv \mathsf{I}$$

that is, the identity function again.

We should be careful when performing substitutions to avoid mixing up free occurrences of an identifier with bound ones. In the expression

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

the function to the left contains a bound y, whereas the y at the right is free. An incorrect substitution would mix the two identifiers in the erroneous result

$$(\lambda y.yy)$$

Simply by renaming the bound y to t we obtain

$$(\lambda x.(\lambda t.xt))y = (\lambda t.yt)$$

which is a completely different result but nevertheless the correct one.

Therefore, if the function $\lambda x.$ <exp> is applied to E, we substitute all *free* occurrences of x in <exp> with E. If the substitution would bring a free variable of E in an expression where this variable occurs bound, we rename the bound variable before performing the substitution. For example, in the expression

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

we associate the argument x with y. In the body

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

only the first x is free and can be substituted. Before substituting though, we have to rename the variable y to avoid mixing its bound with is free occurrence:

$$[y/x](\lambda t.(x(\lambda x.xt))) = (\lambda t.(y(\lambda x.xt)))$$

In normal order reduction we try to reduce always the left most expression of a series of applications. We continue until no further reductions are possible.

2 Arithmetic

We expect from a programming language that it should be capable of doing arithmetical calculations. Numbers can be represented in lambda calculus starting from zero and writing "suc(zero)" to represent 1, "suc(suc(zero))" to represent 2, and so on. In the lambda calculus we can only define new functions. Numbers will be defined as functions using the following approach: zero can be defined as

$$\lambda s.(\lambda z.z)$$

This is a function of two arguments s and z. We will abbreviate such expressions with more than one argument as

$$\lambda sz.z$$

It is understood here that s is the first argument to be substituted during the evaluation and z the second. Using this notation, the first natural numbers can be defined as

$$\begin{array}{lcl} 1 & \equiv & \lambda sz.s(z) \\ 2 & \equiv & \lambda sz.s(s(z)) \\ 3 & \equiv & \lambda sz.s(s(s(z))) \end{array}$$

and so on.

Our first interesting function is the successor function. This can be defined as

$$\mathsf{S} \equiv \lambda wyx.y(wyx)$$

The successor function applied to our representation for zero yields

$$\mathsf{S0} \equiv (\lambda wyx.y(wyx))(\lambda sz.z)$$

In the body of the first expression we substitute all occurrences of w with $(\lambda sz.z)$ and this yields

$$\lambda yx.y((\lambda sz.z)yx) = \lambda yx.y((\lambda z.z)x) = \lambda yx.y(x) \equiv \mathsf{1}$$

That is, we obtain the representation of the number 1 (remember that variable names are "dummies").

Successor applied to 1 yields:

$$\mathsf{S1} \equiv (\lambda wyx.y(wyx))(\lambda sz.s(z)) = \lambda yx.y((\lambda sz.s(z))yx) = \lambda yx.y(y(x)) \equiv \mathsf{2}$$

Notice that the only purpose of applying the number $(\lambda sz.s(z))$ to the arguments y and x is to "rename" the variables used in the definition of our number.

2.1 Addition

Addition can be obtained immediately by noting that the body sz of our definition of the number 1, for example, can be interpreted as the application of the function s on z. If we want to add say 2 and 3, we just apply the successor function two times to 3.

Let us try the following in order to compute 2+3:

$$\mathsf{2S3} \equiv (\lambda sz.s(sz))(\lambda wyx.y(wyx))(\lambda uv.u(u(uv)))$$

The first expression on the right side is a 2, the second is the successor function, the third is a 3 (we have renamed the variables for clarity). The expression above reduces to

$$(\lambda wyx.y((wy)x))((\lambda wyx.y((wy)x))(\lambda uv.u(u(uv)))) \equiv \mathsf{SS3}$$

The reader can verify that $\mathsf{SS3}$ reduces to $\mathsf{S4} = \mathsf{5}$.

2.2 Multiplication

The multiplication of two numbers x and y can be computed using the following function:

$$(\lambda xyz.x(yz))$$

The product of 2 by 2 is then:

$$(\lambda xyz.x(yz))\mathsf{22}$$

which reduces to

$$(\lambda z.\mathsf{2}(\mathsf{2}z))$$

The reader can verify that by further reducing this expression, we can obtain the expected result 4.

3 Conditionals

We introduce the following two functions which we call the values "true"

$$\mathsf{T} \equiv \lambda xy.x$$

and "false"

$$\mathsf{F} \equiv \lambda xy.y$$

The first function takes two arguments and returns the first one, the second function returns the second of two arguments.

3.1 Logical operations

It is now possible to define logical operations using this representation of the truth values.
The AND function of two arguments can be defined as

$$\wedge \equiv \lambda xy.xy(\lambda uv.v) \equiv \lambda xy.xy\mathsf{F}$$

The OR function of two arguments can be defined as

$$\vee \equiv \lambda xy.x(\lambda uv.u)y \equiv \lambda xy.x\mathsf{T}y$$

Negation of one argument can be defined as

$$\neg \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a) \equiv \lambda x.x\mathsf{FT}$$

The negation function applied to "true" is

$$\neg\mathsf{T} \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a)(\lambda cd.c)$$

which reduces to

$$\mathsf{TFT} \equiv (\lambda cd.c)(\lambda uv.v)(\lambda ab.a) = (\lambda uv.v) \equiv \mathsf{F}$$

that is, the truth value "false".

3.2 A conditional test

It is very convenient in a programming language to have a function which is true if a number is zero and false otherwise. The following function Z complies with this requirement

$$\mathsf{Z} \equiv \lambda x.x\mathsf{F}\neg\mathsf{F}$$

To understand how this function works, note that

$$\mathsf{0}fa \equiv (\lambda sz.z)fa = a$$

that is, the function f applied zero times to the argument a yields a. On the other hand, F applied to any argument yields the identity function

$$\mathsf{F}a \equiv (\lambda xy.y)a = \lambda y.y \equiv \mathsf{I}$$

We can now test if the function Z works correctly. The function applied to zero yields

$$\mathsf{Z0} \equiv (\lambda x.x\mathsf{F}\neg\mathsf{F})\mathsf{0} = \mathsf{0F}\neg\mathsf{F} = \neg\mathsf{F} = \mathsf{T}$$

because F applied $\mathsf{0}$ times to $\neg$ yields $\neg$. The function Z applied to any other number N yields

$$\mathsf{ZN} \equiv (\lambda x.x\mathsf{F}\neg\mathsf{F})\mathsf{N} = \mathsf{NF}\neg\mathsf{F}$$

The function F is then applied N times to $\neg$. But F applied to anything is the identity, so that the above expression reduces for any number N greater than zero to

$$\mathsf{IF} = \mathsf{F}$$

3.3 The predecessor function

We can now define the predecessor function combining some of the functions introduced above. When looking for the predecessor of n, the general strategy will be to create a pair $(n, n-1)$ and then pick the second element of the pair as the result.

A pair (a, b) can be represented in λ-calculus using the function

$$(\lambda z.zab)$$

We can extract the first element of the pair from the expression applying this function to T

$$(\lambda z.zab)\mathsf{T} = \mathsf{T}ab = a$$

and the second applying the function to F

$$(\lambda z.zab)\mathsf{F} = \mathsf{F}ab = b$$

The following function generates from the pair $(n, n-1)$ (which is the argument p in the function) the pair $(n+1, n-1)$:

$$\Phi \equiv (\lambda pz.z(\mathsf{S}(p\mathsf{T}))(p\mathsf{T}))$$

The subexpression $p\mathsf{T}$ extracts the first element from the pair p. A new pair is formed using this element, which is incremented for the first position of the new pair and just copied for the second position of the new pair.

The predecessor of a number n is obtained by applying n times the function Φ to the pair $(\lambda.z\mathsf{00})$ and then selecting the second member of the new pair:

$$\mathsf{P} \equiv (\lambda n.n\Phi(\lambda z.z\mathsf{00})\mathsf{F}$$

Notice that using this approach the predecessor of zero is zero. This property is useful for the definition of other functions.

3.4 Equality and inequalities

With the predecessor function as the building block, we can now define a function which tests if a number x is greater than or equal to a number y:

$$\mathsf{G} \equiv (\lambda xy.\mathsf{Z}(x\mathsf{P}y))$$

If the predecessor function applied x times to y yields zero, then it is true that $x \geq y$.

If $x \geq y$ and $y \geq x$, then $x = y$. This leads to the following definition of the function E which tests if two numbers are equal:

$$\mathsf{E} \equiv (\lambda xy. \wedge (\mathsf{Z}(x\mathsf{P}y))(\mathsf{Z}(y\mathsf{P}x)))$$

In a similar manner we can define functions to test whether $x > y$, $x < y$ or $x \neq y$.

4 Recursion

Recursive functions can be defined in the λ calculus using a function which calls a function y and then regenerates itself. This can be better understood by considering the following function Y:

$$\mathsf{Y} \equiv (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$$

This function applied to a function R yields:

$$\mathsf{YR} = (\lambda x.\mathsf{R}(xx))(\lambda x.\mathsf{R}(xx))$$

which further reduced yields:

$$R((\lambda x.\mathsf{R}(xx))(\lambda x.\mathsf{R}(xx))))$$

but this means that $\mathsf{YR} = \mathsf{R}(\mathsf{YR})$, that is, the function R is evaluated using the recursive call YR as the first argument.

Assume, for example, that we want to define a function which adds up the first n natural numbers. We can use a recursive definition, since $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$. Let us use the following definition for R:

$$\mathsf{R} \equiv (\lambda rn.\mathsf{Z}n\mathsf{0}(n\mathsf{S}(r(\mathsf{P}n))))$$

This definition tells us that the number n is tested: if it is zero the result of the sum is zero. If n is not zero, then the successor function is applied n times to the recursive call (the argument r) of the function applied to the predecessor of n.

How do we know that r in the expression above is the recursive call to R, since functions in λ calculus do not have names? We do not know and that is precisely why we have to use the recursion operator Y. Assume for example that we want to add the numbers from $\mathsf{0}$ to $\mathsf{3}$. The necessary operations are performed by the call:

$$\mathsf{YR3} = \mathsf{R}(\mathsf{YR})\mathsf{3} = \mathsf{Z30}(\mathsf{3S}(\mathsf{YR}(\mathsf{P3})))$$

Since $\mathsf{3}$ is not equal to zero, the evaluation reduces to

$$\mathsf{3S}(\mathsf{YR2})$$

that is, the sum of the numbers from $\mathsf{0}$ to $\mathsf{3}$ is equal to $\mathsf{3}$ plus the sum of the numbers from $\mathsf{0}$ to $\mathsf{2}$. Successive recursive evaluations of YR will lead to the correct final result.

Notice that in the function defined above the recursion will be broken when the argument becomes $\mathsf{0}$. The final result will be

$$\mathsf{3S2S1S0}$$

that is, the number $\mathsf{6}$.

5 Projects for the reader

1. Define the functions "less than" and "greater than" of two numerical arguments.
2. Define the positive and negative integers using pairs of natural numbers.
3. Define addition and subtraction of integers.
4. Define the division of positive integers recursively.
5. Define the function $n! = n \cdot (n-1) \cdots 1$ recursively.
6. Define the rational numbers as pairs of integers.
7. Define functions for the addition, subtraction, multiplication and division of rationals.
8. Define a data structure to represent a list of numbers.
9. Define a function which extracts the first element from a list.
10. Define a recursive function which counts the number of elements in a list.
11. Can you simulate a Turing machine using λ calculus?

References

[1] P. M, Kogge, *The Architecture of Symbolic Computers*, McGraw-Hill, New York, 1991, chapter 4.

[2] G. Michaelson, *An Introduction to Functional Programming through Lambda Calculus*, Addison-Wesley, Wokingham, 1988.

[3] G. Revesz, *Lambda-Calculus Combinators and Functional Programming*, Cambridge University Press, Cambridge, 1988, chapters 1–3.