

The untyped Lambda Calculus

Manuel Eberl
eberlm@cs.tum.edu

August 21, 2011

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 2 |
| 2 | Basics | 2 |
| 2.1 | Notation | 2 |
| 2.2 | Examples for abstraction and application | 3 |
| 2.3 | Conversion of expressionsn | 3 |
| 2.3.1 | Free and bound variables | 3 |
| 2.3.2 | Substitution | 4 |
| 2.3.3 | α conversion | 4 |
| 2.3.4 | β reduction | 5 |
| 2.3.5 | β normal form | 5 |
| 2.4 | “non-terminating” expressions | 5 |
| 2.5 | Currying | 6 |
| 3 | Encoding data types | 7 |
| 3.1 | Booleans | 7 |
| 3.2 | Pairs | 7 |
| 3.3 | Natural numbers - the Church numerals | 8 |
| 3.4 | Arithmetic operations | 8 |
| 3.4.1 | Successor | 8 |
| 3.4.2 | Addition | 9 |
| 3.4.3 | Multiplication | 9 |
| 3.4.4 | Predecessor and subtraction | 9 |
| 3.4.5 | Comparisons | 10 |
| 4 | Recursions | 10 |
| 5 | Conclusion | 11 |

1 Introduction

In the early 20th century, several mathematicians tried to define basic, abstract systems upon which one can build all of mathematics. In contrast to systems such as Zermelo-Fraenkel, which is based on sets as the most basic structure, some systems based on functions were developed. One of them is the Lambda Calculus.

The Lambda Calculus was developed by Alonzo Church in 1928 and its first version was published in 1932. In the following years, inconsistencies were found in both the original version of Church's logic in 1932 and in the revised version. (Kleene-Rosser paradoxon) In 1936, Church reduced his logic to what he called "pure Lambda Calculus", the very core of his logic, which is known today as Lambda Calculus. Similarly to older approaches by Peano, Schönfinkel and Curry, functions are defined and applied to one another in the Lambda Calculus. The potential of this simple system was underestimated at first, but it became apparent that the Lambda Calculus is as powerful as the Turing Machine and therefore all computable problems can be expressed in Lambda calculus.

The interest in the Lambda Calculus and Curry's related Combinatoric Logic was initially very low for several years. In the 1960s, however, interest among computer scientists grew giving rise to functional programming languages such as LISP and Haskell.

This paper shall give a short introduction to the untyped Lambda Calculus.

2 Basics

2.1 Notation

The basic idea of the Lambda Calculus is the following: in one single expression, (anonymous) functions are defined and applied to one another. Each Lambda expression is a function mapping a Lambda expression to a Lambda expression.

There are three basic types of expressions which can be nested: *Variables*, *function abstraction* and *function application*. They form the *Lambda expressions*.

Definition 1 (Lambda expression):

Variable A Variable x is a Lambda expression.

Abstraction Let x be a variable and U be a Lambda expression. The abstraction $\lambda x.U$ is a Lambda expression.

Application Let U, V be Lambda expressions. The application $U V$ is a Lambda expression.

Additionally, parentheses may be used for grouping. Without parentheses, the following priority rules apply:

1. Abstractions reach as far to the right as they can, i.e. $\lambda x.a b c$ is equivalent to $\lambda x.(a b c)$, not to $(\lambda x.a) b c$.
2. The application is left-associative, i.e. $U V W$ is equivalent to $(U V) W$, not to $U (V W)$. Therefore, $U V W$ means: "U is applied to V and the resulting function is applied to W", while $U (V W)$ is a simple chaining of functions and means "U is applied to the result of V W".

As a notational simplification, nested abstractions can be condensed, e.g. $\lambda x.\lambda y.T$ may be written as $\lambda xy.T$ for better readability.

The meaning of the expressions is the following:

An *abstraction* $\lambda x.U$ defines an anonymous function $x \rightarrow U$ where U is an expression which may contain

x (then the parameter of the function is plugged in for x), but does not have to contain it. (the function is constant in that case) An “application” applies a function to a parameter.

2.2 Examples for abstraction and application

In the following section, some simple functions are defined and applied in order to demonstrate abstraction and application and also to illustrate the application of one function to another function. (higher order functions)

Examples of simple functions

We defined the following functions:

- $id := \lambda x. x$ - expects one parameter and returns it (identity function)
- $twice := \lambda f x. f (f x)$ - expects a function and another parameter and applies the function to the parameter twice.
- sqr - expects one parameter and squares it¹

These functions can be applied to parameters as shown below.

- $id\ a$: yields a
- $twice\ id\ a$: yields $id\ (id\ a)$, which in turn yields a
- $id\ id$: yields id
- $sqr\ 3$: yields 9
- $twice\ sqr\ 3$: yields 81
- $twice\ sqr$: yields a “to the 4th power” function

Note: there is no way to define identifiers for function - the numbers used above are no symbolic identifiers but only shorthands. In reality, a lambda expression is nothing more than a long sequence of variables, abstractions and applications - one has to replace each of this identifiers with the complete, “expanded” form of the function it represents. Functions cannot be accessed by name. This will be significant when handling recursion. For an illustration of what a Lambda expression without shorthand looks like, see the example at the end of this paper.

2.3 Conversion of expressions

2.3.1 Free and bound variables

Similarly to quantors like \exists and \forall , there are two kinds of variables in Lambda calculus: free and bound. Variables that are defined in a Lambda abstraction are called *bound* within this abstraction, variables that are not bound are called *free*.

Examples

- $\lambda x. x\ y$: x is bound in the abstraction, y is free
- $\lambda x. \lambda x. x$: the x is bound by the inner abstraction and “hides” the outer x
- $(\lambda x. x)\ x$: the first x is bound, the second one is free
- $(\lambda x. \underline{x}) (\lambda y. \underline{x})$: the first underlined x is bound, the second underlined x is free.

¹Not explicitly defined here since encoding numbers in Lambda calculus is explained in a later section. For now, let us simply assume that we can define numbers and apply arithmetic operations such as sqr to them.

A more formal definition of free and bound variable is the following:

Definition 2 (Free variables):

Let $FV(A)$ be the set of free variables in an expression A . This set is defined recursively:

- $FV(x) = \{x\}$ for a variable x
- $FV(\lambda x. B) = FV(B) \setminus \{x\}$ for a Lambda expression B and a variable x
- $FV(B C) = FV(B) \cup FV(C)$ for Lambda expressions B, C

A Lambda expression without free variables is called *closed Lambda expression* or *combinator*.

2.3.2 Substitution

An important operation for further conversions is the substitution, i.e. replacing a variable with an expression.

Definition 3 (Substitution):

The substitution $[x := T]$ of x with T is defined as:

- $x[x := R] = R$
- $a[x := R] = a$ (for $x \neq a$)
- $(S T)[x := R] = S[x := R] T[x := R]$
- $(\lambda x. T)[x := R] = \lambda x. T$ (no substitution since x is bound in this case)
- $(\lambda a. T)[x := R] = \lambda a. T[x := R]$ (for $x \neq a$ and a does not appear as a free variable in R , “freshness condition”)

The last condition (a does not appear as a free variable in R) is very important. Another way to express this is: “ a must be fresh in R ”, because substitutions can lead to problems otherwise - when R is plugged in, the a in R are not free anymore but are interpreted as the bound a . (“capturing”)

Example: Let $A = \lambda a. a \text{ narf } foo$. We want to evaluate $A[foo := bar \ a \ wuppd]$. To do that, we have to replace all occurrences of foo with the expression $bar \ a \ wuppd$. However, a is not fresh in $bar \ a \ wuppd$, as the expression contains it as a free variable.

A simple substitution would result in the expression $\lambda a. a \text{ narf } (bar \ a \ wuppd)$ in which the a in parentheses is the bound a - not the free one that it should be. This is why the freshness condition is important. The current definition of substitution is only partial. When the freshness condition is not satisfied, the a in the abstraction must be renamed into a fresh variable with α conversion to avoid capturing. With this addition to the definition, substitution is total.

2.3.3 α conversion

As it is the case with quantors, the names of bound variables are irrelevant. The Lambda expressions $\lambda x. x$ and $\lambda y. y$ are completely equivalent as they yield the same results for all parameters. The process of renaming a bound variable in an abstraction is called *α conversion*.

Definition 4 (α conversion):

The α conversion of an abstraction $\lambda x.S$ for variables x,y and a Lambda expression S where y is fresh in S is defined as:

$$\lambda x.S \stackrel{\alpha}{\equiv} \lambda y.S[x := y]$$

Expressions that can be converted to one another through α conversion are called α -equivalent.

The condition that y is fresh in S ensures that we do not need another α conversion to resolve the problem. With that definition, we can solve the above problem of the substitution:

$$\begin{aligned} & (\lambda a.a \text{ narf } foo)[foo := bar \ a \ wuppd\ i] \stackrel{\alpha}{\equiv} \\ & \stackrel{\alpha}{\equiv} (\lambda a'.a' \text{ narf } foo)[foo := bar \ a \ wuppd\ i] = \lambda a'.a' \text{ narf } (bar \ a \ wuppd\ i) \end{aligned}$$

2.3.4 β reduction

β reduction is a resolution of an application. For this purpose, we first define the term *redex* (reducible expression):

Definition 5 (Redex):

An expression of the form $(\lambda x.S) T$ where x is a variable and S, T are Lambda expressions is called *Redex* (reducible expression). An expression that does not contain any redexes is called *irreducible*.

The β reduction is an operation that transforms a redex as defined above by plugging in the parameter T into S for x . This can be expressed by substitution:

Definition 6 (β reduction):

The β reduction of a redex $(\lambda x.S) T$ is defined as:

$$(\lambda x.S) T \stackrel{\beta}{\Rightarrow} S[x := T]$$

2.3.5 β normal form

With many Lambda expressions, an irreducible expression can be reached after a finite number of β reductions. This irreducible expression is called β normal form. An interesting property of the β reduction is its *confluence*, i.e. if an expression A has a normal form, all possible irreducible expressions that can be reached through β reduction of A are α -equivalent to the normal form; this means that there is at most one β reduction, irrespective of the order in which the reductions are performed.

However, it is sometimes possible to reduce an expression with a normal form an infinite amount of times without ever obtaining a normal form (see next section) if a bad reduction strategy is chosen. But whenever two different reduction processes lead to a normal form, the results are guaranteed to be the same. (save α conversion)

2.4 “non-terminating” expressions

Since the Lambda calculus is Turing-complete, there have to be expressions that do not “terminate”. This means that they do not have a β normal form - no irreducible expression can be obtained through a finite number of reductions. The expression may even grow indefinitely as reduction progresses. Because of the Turing-completeness, the question whether a Lambda expression has a normal form is undecidable as it is equivalent to the halting problem.

A textbook example of an expression without a β normal form is the Ω combinator:

$$\omega := \lambda x. x x$$

$$\Omega := \omega \omega = (\lambda x. x x) (\lambda x. x x)$$

The β reduction of Ω results in Ω again. Therefore, Ω is invariant under reduction. Ω can also be modified to grow indefinitely under reduction:

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

Reduction leads to:

$$(\lambda x. x x x) (\lambda x. x x x) \Rightarrow (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) \Rightarrow (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) \Rightarrow \dots$$

An example for an expression that has a normal form but also has a reduction strategy that leads to an infinite amount of reductions without ever reaching the normal form as stated in the previous section is the following expression:

$$false \Omega id = (\underline{\lambda ab. b}) ((\underline{\lambda x. x x}) (\lambda x. x x)) (\lambda x. x)$$

If the redex with the first underlined abstraction is reduced, the entire expression evaluates to $id = \lambda x. x$ in one step. If, however, the redex with the second underlined abstraction is reduced, the expression does not change at all, and this can be repeated infinitely. Therefore, while the order of reduction does not change the result, it can indeed determine whether a result is found in the first place.

Recursive functions in particular must be evaluated in a “lazy” way, i.e. only results that are actually required for the evaluation are calculated, as recursive expression would never terminate otherwise.

2.5 Currying

As stated previously, a Lambda expression is a function that maps a Lambda expression to a Lambda expression. Functions with more than one parameter do not exist in the Lambda calculus - it is, however, possible to build these functions indirectly. If one wants to define a function such as *add* that expects two parameters a and b and adds them, one can interpret *add* as a function of a that returns another function add_a , which is effectively an “increase by a ” function, expecting one parameter b and returning $a + b$. The expression *add a b* reduces to $add_a b$, which in turn reduces to $a + b$.

add 4, for instance, yields an “increase by 4” function. *add 4 38* is an application of this “increase by 4” function to 38 and the resulting expression reduces to 42.

This can be done for every function f with n parameters: convert f to a function mapping the first parameter a_1 to a function f'_{a_1} . This function in turn maps the second parameter a_2 to f'_{a_1, a_2} and so on, until a function $f'_{a_1, a_2, \dots, a_{n-1}}$ is reached, which maps the last parameter a_n to the result $f'_{a_1, a_2, \dots, a_{n-1}, a_n} = f(a_1, a_2, \dots, a_{n-1}, a_n)$. This is called *Currying* and it is possible because $A \times A \times \dots \times A$ is isomorphic to $A \rightarrow (A \rightarrow (A \rightarrow \dots A))$

Definition 7 (Functions with more than one parameter - Currying):

Let f be a function of n parameters:

$$f : A_1 \times A_2 \times \dots \times A_n \rightarrow B$$

This function can be converted to a higher order function in the following way:

$$f' : A_1 \rightarrow (A_2 \rightarrow (\dots (A_n \rightarrow B)))$$

Example:

Let *reverse* be a function that takes two functions f and g as parameters and returns $g f$ - i.e. applies g to f .

$$reverse := \lambda f g. g f$$

When we apply *reverse* to two parameters a and b , we get:

$$\text{reverse } a \ b = (\lambda f g. g \ f) \ a \ b \Rightarrow (\lambda g. g \ a) \ b = \text{reverse}'_a \ b \Rightarrow b \ a$$

The first reduction step yields a function $\text{reverse}'_a$ which expects a parameter g and applies it to a . The second reduction step then yields the result $b \ a$.

3 Encoding data types

The only structures that exist in the Lambda calculus are variables and functions. One might ask how common data types such as booleans and numbers or more complex structures such as lists can be built using only functions. To do this, functions have to be defined with certain behaviour and operations on these data types are realised by applying functions to them. The easiest data type that can be expressed that way are Booleans.

3.1 Booleans

There are two Boolean values - *true* and *false* - and four “interesting” operations on them - the unary *not* and the ternary *and*, *or*, *xor*. One way to encode *true* and *false* is as a function that takes two parameters and returns one of them. *true* returns the first one, *false* returns the second one. The definition of *true* and *false* is therefore very simple and intuitive:

Definition 8 (*true* and *false*):

$\text{true} := \lambda ab. a$

$\text{false} := \lambda ab. b$

With that, “if-then-else” branches can be realised: consider the expression $a \ U \ V$ where a is a Boolean as defined above and U and V are Lambda expressions. If $a = \text{true}$, V is ignored and the expression reduces to U . If $a = \text{false}$, U is ignored and the expression reduces to V . Therefore, the expression $a \ U \ V$ is effectively a branch **if a then U else V** ;

Using this “if” branch, all Boolean operators can be defined very intuitively. A *not* is simply **if a then false else true** ;

A *and* can be expressed similarly: if a and b should be calculated, one can first look at a . If $a = \text{false}$, the result is *false* irrespective of b . If $a = \text{true}$, the result only depends on b , as the result is *true* iff b is *true*. Therefore, *and* is nothing more than **if a then b else false** ;

We can now define all four important Boolean operations in the Lambda calculus:

| operator | if-then-else | Lambda expression |
|------------|---|---|
| <i>not</i> | if a then false else true ; | $\lambda a. a \ \text{false} \ \text{true}$ |
| <i>and</i> | if a then b else false ; | $\lambda ab. a \ b \ \text{false}$ |
| <i>or</i> | if a then true else b ; | $\lambda ab. a \ \text{true} \ b$ |
| <i>xor</i> | if a then $\text{not } b$ else b ; | $\lambda ab. a \ (\text{not } b) \ b$ |

3.2 Pairs

Encoding pairs (2-tuples) is also very intuitive. What properties is a pair supposed to have? It has to contain two Lambda expressions and they can be accessed. One possible solution to encode this behaviour in function is the following:

A pair is a function that expects another function z as a parameter and applies z to its two values. This means that a pair $p = (a, b)$ maps another function z to $z \ a \ b$.

How can the values a and b be accessed? This is done by two functions called *first* and *second*, which expect a pair $p = (a; b)$ and return a or b , respectively. In order to do that, we have to apply p to a function which returns its first or second parameter, respectively. We have already defined such functions: *true* and *false*. The expression $p \text{ true}$ evaluates to a and the expression $p \text{ false}$ evaluates to b .

Definition 9 (Pairs):

A pair $p = (a; b)$ is encoded as $\lambda z. z a b$.

Access to pairs is done with the following two functions:

$\text{first} := \lambda p. p \text{ true}$

$\text{second} := \lambda p. p \text{ false}$

3.3 Natural numbers - the Church numerals

Encoding natural numbers is more difficult. Church used the Church numerals, which were named after him: a natural number n is encoded as a function which maps another function f to f^n , i.e. that chains f with itself n times², i.e. $n : f x \rightarrow f^n x$, or, as a Lambda expression:

Definition 10 (Church numerals):

The Church numeral encoding $n \in \mathbb{N}_0$ is:

$\lambda f x. \underbrace{f (f \dots (f x) \dots)}_{n \text{ mal}} = \lambda f x. f^n x$

Examples:

$0 \hat{=} \lambda f x. x$ (maps every function to *id* and is interestingly enough equivalent to *false*)

$1 \hat{=} \lambda f x. f x$

$2 \hat{=} \lambda f x. f (f x)$

...

3.4 Arithmetic operations

3.4.1 Successor

The most basic operation for working with natural numbers is the successor operation $\text{succ} : n \rightarrow n + 1$. This can be done relatively easily in Lambda calculus: since n is a function that maps f and x to $f^n x$, the successor of n has to be a function that maps f and x to $f^{n+1} x$. Therefore, we can calculate $f^n x$, which is $n x$ and apply f to the result once more.

Definition 11 (Successor - succ):

$\text{succ} := \lambda n f x. f (n f x)$

Example:³

$$\begin{aligned} \text{succ } 1 &= \text{succ } (\lambda f x. f x) = (\lambda n f x. f (n f x)) (\lambda f x. f x) \Rightarrow \\ &\Rightarrow \lambda f x. f ((\lambda f' x'. f' x') f x) \Rightarrow \\ &\Rightarrow \lambda f x. f (f x) = 2 \end{aligned}$$

²Therefore, the Church numeral 2 is equivalent to the *twice* function we used in an example before.

³The first underlined part is always the abstraction reduced in that step, the following underlined parts are the parameters plugged in in the reduction

3.4.2 Addition

The addition $m + n$ can be defined in two different ways: one way is to apply the successor function to n m times, i.e. $\text{succ } n$. However, the resulting expressions get rather lengthy when evaluated by hand, which is why we will use another method.

We first calculate $n \text{ } f \text{ } x$ - which is $f^n x$ - and apply f^m to the result, yielding $f^{m+n} x$.

Definition 12 (Addition):

$$+ := \lambda mnfx. m \text{ } f \text{ } (n \text{ } f \text{ } x)$$

Example:

$$\begin{aligned} + \text{ } 2 \text{ } 3 &= (\lambda mnfx. m \text{ } f \text{ } (n \text{ } f \text{ } x)) \text{ } 2 \text{ } 3 \Rightarrow \\ &\Rightarrow \lambda fx. 2 \text{ } f \text{ } (3 \text{ } f \text{ } x) = \\ &= \lambda fx. (\lambda f'x'. f' \text{ } (f' \text{ } x')) \text{ } f \text{ } ((\lambda f'x'. f' \text{ } (f' \text{ } (f' \text{ } x')))) \text{ } f \text{ } x) \Rightarrow \\ &\Rightarrow \lambda fx. (\lambda x'. f \text{ } (f \text{ } x')) \text{ } ((\lambda f'x'. f' \text{ } (f' \text{ } (f' \text{ } x')))) \text{ } f \text{ } x) \Rightarrow \\ &\Rightarrow \lambda fx. (\lambda x'. f \text{ } (f \text{ } x')) \text{ } (f \text{ } (f \text{ } (f \text{ } x))) \Rightarrow \\ &\Rightarrow \lambda fx. f \text{ } (f \text{ } (f \text{ } (f \text{ } (f \text{ } x)))) = 5 \end{aligned}$$

3.4.3 Multiplication

In order to calculate multiplication, we consider the function $m \text{ } f$. This function expects one parameter x and returns $f^m x$. Simply put, the expanded expression contains a sequence of $m \text{ } f$'s. So what happens when n is applied to that function? The function is chained n times, i.e. we now have a n sequences of $m \text{ } f$'s in a row, so this function maps x to $f^{m \cdot n} x$, which is effectively a multiplication. Therefore, we simply have to calculate $n \text{ } (m \text{ } f)$ or analogously $m \text{ } (n \text{ } f)$.

Definition 13 (Multiplication):

$$* := \lambda mnf. m \text{ } (n \text{ } f)$$

3.4.4 Predecessor and subtraction

Finding a predecessor of a Church numeral in Lambda calculus is significantly more difficult than finding the successor, as applying a function once more is simple - but applying it once less is not. Since we have natural numbers, 0 does not have a predecessor. For simplicity, however, we will define $\text{pred } 0 = 0$, or in other words: $\text{pred } n := n \div 1$

\div is a saturated subtraction. i.e. $m \div n = \max(m - n, 0)$. Therefore, if $m < n$, $m \div n = 0$, since we are trying to avoid negative numbers. If $m \geq n$, $m \div n = m - n$.

The Peano axioms demand that the 0 not have a predecessor, but in this context it is more useful to artificially define it as its own predecessor.

To facilitate the calculation of the predecessor of n , we define a helper function Φ , which does the following mapping of a pair $p = (a, b)$ to another pair:

$$\Phi : (a, b) \longrightarrow (a + 1, a)$$

$$\Phi := \lambda pz. z \text{ } (\text{succ } (\text{first } p)) \text{ } (\text{first } p)$$

When this function is applied to $(0,0)$, it returns $(1,0)$. If it is in turn applied to $(1,0)$, it returns $(2,1)$ and so on. Following this pattern, the result of n chained applications of the function to $(0,0)$ is the pair $(n, n \dot{-} 1)$. Therefore, the predecessor we are looking for is the second element of this pair.

Thus, we get:

$$pred := \lambda n. second (n \Phi \lambda z. z 0 0)$$

By analogy to the first variant presented for addition, we can define the (saturated) subtraction $m \dot{-} n$ as a n applications of the predecessor function to m :

$$\dot{-} := \lambda mn. n pred m$$

3.4.5 Comparisons

The most basic comparison is the test whether a number is equal to 0. This will enable the construction of more advanced comparisons later. In order to compare a Church numeral to 0, we use the fact that 0 maps every function f to the identity function, i.e. $(\forall f)[0 f \equiv id]$

We can now construct a helper function h , which takes one parameter x and always returns *false*. When n is applied to h , we can distinguish two cases:

- If $n = 0$, then $n h = id$ and $n h$ returns its parameter.
- If $n \neq 0$, then $n h$ ignores its parameter and returns *false*.

Therefore, if h is chained n times and applied to *true*, we obtain an *iszero* function.

Definition 14 (Test for 0 - iszero):

$$iszero := \lambda n. n (\lambda x. false) true$$

In order to define our actual comparison operators, we consider the \leq operator and rewrite its definition: $m \leq n \Leftrightarrow m - n \leq 0 \Leftrightarrow m \dot{-} n = 0^4$

Thus, we can reduce \leq to *iszero* and $\dot{-}$. Conversely, we can obtain a \geq operator by swapping the parameters. Negating the \leq and \geq operators yields $>$ and $<$ and combination of \leq and \geq with a Boolean *and*, we can obtain a $=$.

Definition 15 (Comparison operators):

$$\leq := \lambda mn. iszero (\dot{-} m n)$$

$$\geq := \lambda mn. iszero (\dot{-} n m)$$

$$> := \lambda mn. not (\leq m n)$$

$$< := \lambda mn. not (\geq m n)$$

$$= := \lambda mn. and ((\leq m n) (\geq m n))$$

4 Recursions

We have already implicitly introduced a recursion with fixed recursion depth - each application of a Church numeral n to a function f applies f n times recursively to itself. (see e.g. *pred*). However, this is vaguely similar to a “for” loop or primitively recursive approach. (cf. LOOP computability) Recursive functions with a variable termination condition would be more interesting. As an example, we will use the following recursive definition of the factorial $n!$:⁵

$$n! := \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

⁴see definition of the saturated subtraction

⁵The factorial, of course, is also primitively recursive and LOOP-computable, it could be defined in the same way as the predecessor function, with a helper function such as the Φ that was used in *pred*

Superficially, one could construct the following Lambda expression:

$$fac_rec := \lambda n. (iszero\ n)\ 1\ (*\ n\ (fac_rec\ (pred\ n)))$$

This, however, leads to a problem: *fac_rec* is used in its own definition. But at this point, it is not fully defined yet. If one were to try and expand the shorthands in the expression above, one would have to replace *fac_rec* with the entire expression again, yielding an expression that again contains *fac_rec*, which would have to be expanded again and so on. The expanded expression would have infinite length.

The problem, in a nutshell, is, that *fac_rec* does not “know itself” and thus cannot call itself. A simple way to resolve this is to add another parameter to *fac_rec* - namely a function that is assumed to be *fac_rec*. Instead of directly calling *fac_rec* in its definition, it calls this function *f* and passes *f* to it as an additional parameter.

We can now define *fac_rec*:

$$fac_rec := \lambda fn. (iszero\ n)\ 1\ (*\ n\ (f\ f\ (pred\ n)))$$

To obtain a more “convenient” function, we can now define an encapsulating function *fac*, which hides this workaround:

$$fac := \lambda n. fac_rec\ fac_rec\ n$$

Finally, in order to illustrate what a complete, expanded Lambda expression looks like, we shall give an expanded version of *fac* 3 without any shorthands in order to appreciate the complexity that results from these several layers of abstraction that we have defined:

$$\begin{aligned} & (\lambda n. (\lambda fn_1. ((\lambda n_2. n_2 (\lambda x. (\lambda ab. b)) (\lambda ab. a)) n_1) (\lambda fx. f\ x) ((\lambda mn_2f. m\ (n_2\ f)) n_1 (f\ f\ ((\lambda n_3. \\ & (n_3 (\lambda pz. z ((\lambda n_4fx. f\ (n_4\ f\ x))(p\ (\lambda ab. a))) (p\ (\lambda ab. a))) (\lambda z. z (\lambda ab. b) (\lambda ab. b))) (\lambda ab. b)) n_1)))) \\ & (\lambda fn_1. ((\lambda n_2. n_2 (\lambda x. (\lambda ab. b)) (\lambda ab. a)) n_1) (\lambda fx. f\ x) ((\lambda mn_2f. m\ (n_2\ f)) n_1 (f\ f\ ((\lambda n_3. (n_3 (\lambda pz. z \\ & ((\lambda n_4fx. f\ (n_4\ f\ x))(p\ (\lambda ab. a))) (p\ (\lambda ab. a))) (\lambda z. z (\lambda ab. b) (\lambda ab. b))) (\lambda ab. b)) n_1)))) n) \lambda fx. f\ (f\ (f\ x)) \end{aligned}$$

This expression evaluates to:

$$\lambda fx. f\ (f\ (f\ (f\ (f\ (f\ x)))) = 6$$

5 Conclusion

The untyped Lambda calculus is, just like the Turing machine, a minimal, abstract model for computability and therefore a useful tool for proofs in theoretical computer science. However, it is completely unsuitable for actual programming. The biggest problem is the lack of typing - a function for Booleans can be applied to Church numerals or even arithmetic operations. When this happens by accident, the mistakes can be difficult to trace. In general, calculations in typed Lambda calculus are unnecessarily convoluted, intricate and impractical. Just like the Turing machine it is merely an abstract model; applying it for actual computations is not very sensible.

This problems can be solved by introducing native types for Booleans, numbers, lists and so on instead of using the rather pedestrian encodings. If some other modifications are applied an “syntactic sugar” is added to make the calculus more usable, the result is a complete functional programming language such as LISP or Haskell. Because of their tremendously useful properties, Lambda expressions and derived functional concepts have also been introduced to imperative and object-oriented languages such as Python, Scala and - indirectly - even C++.