Introduction into the (un)typed Lambda calculus

Kasper Engelen

Abstract. This paper serves as an introduction into the lambda calculus. It begins with an overview of the untyped lambda calculus. The different notations, operations, and encodings, such as abstraction, application, alpha conversion, beta reductions, Church numerals, arithmetic, etc. will be explained. Next, the paper will give an introduction into the simply typed lambda calculus. At each stage it will also pay attention to what one can and cannot do with the lambda calculus in its different forms. The concepts of Turing-completeness, decidability and the equivalence to other models of computability will be explored where appropriate.

Keywords— lambda calculus, typing, logic, decidability, functions

Contents

1 Untyped λ-calculus

In this section we will explore the untyped λ-calculus, this is the original version of the calculus and was first introduced by Church 1932. [9]

1.1 Syntactic constructs

The lambda calculus has a very simple syntax that consists of the following three basic expressions:

- "x", with x being a variable (variable expressions),
- "MN", with M and N being expressions (application), and
- "$\lambda x.M$" with x being a variable and M being an expression (abstraction). [10]

We can also state this in a more formal way. Namely, we will define the set of λ-expressions. We shall refer to this set as Λ. We will assume that V is the set of all variables.

$$x \in V \implies x \in \Lambda, \tag{1}$$
$$M, N \in \Lambda \implies (MN) \in \Lambda, \tag{2}$$
$$M \in \Lambda, x \in V \implies (\lambda x.M) \in \Lambda. \tag{3}$$

Here, (1) refers to variable expressions, (2) refers to application, and (3) refers to abstraction. [4]

Parentheses can also be used to group certain expressions together. By default, the order of operations is as follows:

- Abstraction has the highest order and unless limited by parentheses, abstractions reach as far to the right as possible: $\lambda x.a\,b\,c$ is equal to $\lambda x.(a\,b\,c)$. [10]
- Application is left-associative: $U\,V\,W$ is equal to $(U\,V)\,W$. [10]

In this paper we will refer to the set of variables that are part of a λ-term M as $V(M)$. It is defined as follows: [12]

$$V(x) = \{x\}$$
$$V(M\,N) = V(M) \cup V(N)$$
$$V(\lambda x.M) = V(M) \cup V(x)$$

1.2 Abstraction and application

The abstraction is the way functions are specified in the lambda calculus. Roughly speaking, "$\lambda x.x^2$" can be said to be equivalent to "$f(x) = x^2$". Application on the other hand, is the way functions are applied to arguments in the lambda calculus, we can again say that "$f\,2$" is equivalent to "$f(2)$". [5]

All the functions specified using the abstraction syntax are *anonymous* i.e. they have no identifier. This is quite similar to how *lambda functions* are specified in C++ and Python. Later on we will see that this is a problem when we want to define recursion, but this can be overcome by using some clever notations. [10]

Abstraction is a *binding construct* and as such introduces the concepts of *bound* and *unbound* variables. The bound variables are those that appear in the abstraction construct i.e. all variables that are specified as a parameter of a function. We can compare this to the variables in integral calculus. Take for example $\int f(x, y)\,\mathrm{d}x$. Here x is "bound" while y is "free". [4] [11]

Again, we can define these concepts in a formal way. We let $FV(A)$ be the set of all the free variables in an expression A. We define the set $FV(A)$ recursively in the following manner:

- $FV(x) = \{x\}$, where x is a variable expression,
- $FV(\lambda x.M) = FV(M) \setminus \{x\}$, where M is a λ-expression and x is a variable,
- $FV(MN) = FV(M) \cup FV(N)$, where M and N are λ-expressions. [10] [4]

It is also from this that we can derive the definition of a *combinator*. A combinator is simply a λ-term M so that $FV(M) = \emptyset$. A λ-term with no free variables is also called a *closed* term. [4] Combinators play a key role in combinatory logic. [9]

Within the λ-calculus, functions with more than one parameter do not exist, since we apply one function to one argument at a time. It is however possible to create constructs that take more than parameter by "chaining" abstractions. We call this *currying* and this is done as follows:

Let's say we have a function $f : A_1 \times A_2 \times \cdots \times A_n \to B$, or if we want to put this into λ-notation: $\lambda a_1 \cdots a_n.M$. We can then transform this function into $f : A_1 \to (A_2 \to (\cdots (A_n \to B)))$, which would be $\lambda a_1.\lambda a_2.\cdots.\lambda a_n.M$ in λ-notation. Instead of a function with n parameters, we now have n nested functions that take one parameter.

Each time we apply argument a_i to function f_i, we get a function f_{i+1} in return, we continue this application process until all arguments are applied and we end up with M. [4] [10].

We can illustrate this with an example. Let's say we have a function *add*, that takes two arguments a and b and adds them together. After applying *add* to the first argument a, we get add_a in return. This function takes an argument and adds it to a. When we apply add_a to the second argument b, this reduces to $a + b$. [10].

1.3 The substitution mechanism

In order transform λ-expressions into other λ-expressions, we have a mechanism called *substitution*. It is defined recursively as follows: [4] [11]

$$
\begin{aligned}
x[x := N] &\equiv N \\
y[x := N] &\equiv y \\
(P\,Q)[x := N] &\equiv (P[x := N])(Q[x := N]) \\
(\lambda y.P)[x := N] &\equiv \lambda y.(P[x := N]), \text{provided } y \not\equiv x \qquad (1) \\
(\lambda x.P)[x := N] &\equiv \lambda x.(P[x := N]) \qquad (2)
\end{aligned}
$$

Equation (2), and the provision in (1), refer to the fact that during substitution, bound variables are not replaced. Also if one makes sure to choose the variables so that the bound variables differ from the free ones in a term, the provision specified in (1) is not needed. For example, one can use $y(\lambda xy'.xy'z)$ instead of $y(\lambda xy.xyz)$. [4] [12]

An example: [4]

$$yx(\lambda x.x)[x := N] \equiv yN(\lambda x.x)$$

1.4 Alpha

One can see that $\lambda x.x$ and $\lambda y.y$ are basically the same thing. In the lambda calculus these terms are said to be α-*equivalent*, because they can be converted into each other by one or more α-*conversions*. [11] The fact that two λ-terms M and N are α-equivalent can be written as $M \overset{\alpha}{\equiv} N$.

An α-conversion is an operation on a λ-expression in which all occurences of a bound variable, are replaced by another variable that does not already occur in the existing expression. [11] [4]

A formal definition is also possible: [12]

$$\frac{}{x \overset{\alpha}{\equiv} x} \qquad \frac{M \overset{\alpha}{\equiv} A, N \overset{\alpha}{\equiv} B}{M\,N \overset{\alpha}{\equiv} A\,B} \qquad \frac{z \notin V(M) \cup V(N), M[x := z] \overset{\alpha}{\equiv} N[y := z]}{\lambda x.M \overset{\alpha}{\equiv} \lambda y.N}$$

Note: the notation here means that what is below the line, can be inferred from what is above the line.

The property of α-equivalence can also be used to divide a set of λ-expressions into equivalence classes. The elements of each class are said to be equivalent modulo α-equivalence. For example, the set $\{\lambda x.x, \lambda y.y, \lambda xy.z, \lambda x.y\}$ can be devided up into the following equivalence classes: $\{\lambda x.x, \lambda y.y\}$, $\{\lambda xy.z\}$, and $\{\lambda x.y\}$. [11]

1.5 Beta

One of the most important aspects of the theory concerning the λ-calculus, is the β-reduction. It follows from the principal axiom of the lambda calculus

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

and can be used to resolve the application of one λ-term to another. [4]

We can also define the β-reduction as a mathematical relation. First we will formally define some properties about relations: [4]

1. A relation R on Λ is *compatible* iff
$$M\,R\,N \implies (Z\,M)\,R\,(Z\,N),$$
$$(M\,Z)\,R\,(N\,Z), \text{ and}$$
$$(\lambda x.M)\,R\,(\lambda x.N).[4]$$
2. A *congruence* relation on Λ is a compatible equivalence relation. [4] A *congruence* relation is an equivalent relation, so that when an algebraic operation is done on elements that are equivalent, the resulting elements will still be equivalent. [3]
3. A *reduction* relation on Λ is a compatible, reflexive and transitive relation. [4] It is assumed the reader understands the concepts of reflexivity and transitivity.

The relation $\overset{\beta}{\to}$ is a β-reduction in one step and is defined as follows: [4]

1. $(\lambda x.M)N \overset{\beta}{\to} M[x := N]$, ($\beta$-reduction)
2. $M \overset{\beta}{\to} N \implies Z\,M \overset{\beta}{\to} Z\,N, M\,Z \overset{\beta}{\to} N\,Z$, and $\lambda x.M \overset{\beta}{\to} \lambda x.N$. (compatibility)

The relation $\overset{\beta}{\twoheadrightarrow}$ is the reflexive transitive closure of $\overset{\beta}{\rightarrow}$ and is thus a reduction relation. It is defined as follows: [4]

1. $M \overset{\beta}{\twoheadrightarrow} M$, (reflexivity)
2. $M \overset{\beta}{\rightarrow} N \implies M \overset{\beta}{\twoheadrightarrow} N$,
3. $M \overset{\beta}{\twoheadrightarrow} N, N \overset{\beta}{\twoheadrightarrow} L \implies M \overset{\beta}{\twoheadrightarrow} L$. (transitivity)

Note that by analysing these three rules, we can also deduce that $\overset{\beta}{\twoheadrightarrow}$ is compatible.

The relation $\overset{\beta}{\equiv}$ is a congruence relation and is defined as follows: [4]

1. $M \overset{\beta}{\twoheadrightarrow} N \implies M \overset{\beta}{\equiv} N$,
2. $M \overset{\beta}{\equiv} N \implies N \overset{\beta}{\equiv} M$, (symmetry),
3. $M \overset{\beta}{\equiv} N, N \overset{\beta}{\equiv} L \implies M \overset{\beta}{\equiv} L$. (transitivity)

Note that because of the compatibility of $\overset{\beta}{\twoheadrightarrow}$, $\overset{\beta}{\equiv}$ is also compatible.

The β-reduction introduces the concept of a *redex* (**red**ucible **ex**pression). A redex is a λ-expression of the form $(\lambda x.M)N$, where x is a variable , and M and N are λ-expressions. The result of $M[x := N]$ is called the *contractum*. [4] [10]

An expression that does not contain any redexes is called *irreducible*. [10] An irreducible λ-expression is also called a β-normal form. A λ-expression M has a normal form N if $M \overset{\beta}{\equiv} N$. [10]

According to the *Church-Rosser* theorem, all the normal forms of an expression are α-equivalent. This means if two different reduction strategies both lead to a normal form, these normal forms are guaranteed to be α-equivalent. [10] We call this propery *confluence*. [6]

1.6 Eta

Another form of conversion exists in the λ-calculus: the η-conversion. [12] [7]

$$(\lambda x.M\, x) \overset{\eta}{\equiv} M \tag{1}$$

$$(\lambda x.M\, x) \overset{\eta}{\rightarrow} M \tag{2}$$

$$M \overset{\eta}{\rightarrow} (\lambda x.M\, x) \tag{3}$$

The equivalence in (1) is called the η-equivalence. Two terms are η-equivalent if they can be converted into eachother with a η-conversion. There are two types of η-conversion. The first one is described in (2) and is called the η-reduction, the second one is described in (3) and is called the η-abstraction. [7]

It can be shown that the η-equivalence is a compatible relation, and as such is a congruence relation. To do this, we apply the same technique as we did to prove the β-equivalence is compatible.

The η-reduction is useful to eliminate redundant lambda abstractions. Informally this means that if the sole purpose of a lambda abstraction is to pass its argument to another function, then it can be eliminated. [7]

The aim of the η-abstraction is the opposite of that of the η-reduction. Here, we encapsulate a λ-expression in a "redundant" lambda abstraction. This can be useful when applying *eager evaluation*, a technique which is used in certain programming languages, in order to prevent the immediate evaluation of a certain term. [7]

The η-conversion is allowed at any place in a λ-expression, not only at the root of the λ-expression. [12].

1.7 Encodings

Even though the λ-calculus only consists of functions and variables, it is possible to represent more complex structures such as numerals, pairs, and arithmetic. [10]

1.7.1 Booleans

The first concept we will introduce the representation of *booleans*. They are defined as follows:

$$\mathbf{true} := \lambda ab.a$$
$$\mathbf{false} := \lambda ab.b$$

True and false are simply functions that take two arguments and return either the first or the second argument. [10] Note that here we assign these functions the names **true** and **false**, be this is merely a shorthand since in λ-calculus function names don't exist. Later on we will do something similar for the Church numerals.

An if-then-else clause can now be represented by the λ-term $B\,P\,Q$, since [4]

$$\mathbf{true}\,P\,Q \xrightarrow{\beta} P$$
$$\mathbf{false}\,P\,Q \xrightarrow{\beta} Q$$

The following table contains a list of well-known boolean concepts represented as λ-expressions: [10]

operator	if-then-else	λ-expression
not	**if** a **then** false **else** true **;**	$\lambda a.a\,\mathbf{false}\,\mathbf{true}$
and	**if** a **then** b **else** false **;**	$\lambda ab.a\,b\,\mathbf{false}$
or	**if** a **then** true **else** b **;**	$\lambda ab.a\,\mathbf{true}\,b$
xor	**if** a **then** not b **else** b **;**	$\lambda ab.a\,(\mathbf{not}\,b)\,b$

1.7.2 Pairs

Representing pairs can easily be done by a simple abstraction, and they can be accessed by using the previously defined **true** and **false**. We will define a pair (a, b) as the λ-expression $\lambda z.z\,a\,b$ with $a, b \in \Lambda$. We will abbreviate this as $[a, b]$. [4]

Accessing the data contained within the pairs is very simple:

$$[a, b]\,\mathbf{true} \xrightarrow{\beta} a$$
$$[a, b]\,\mathbf{false} \xrightarrow{\beta} b \quad [4].$$

We can also encapsulate **true** and **false** as follows: [10]

$$\mathbf{first} := \lambda p.p\,\mathbf{true}$$
$$\mathbf{second} := \lambda p.p\,\mathbf{false}$$

1.7.3 Natural numbers

In order to represent the natural numbers, we will simply apply a function f to itself n times. As such the number n will be defined as $\lambda fx.f^n\,x$. An example of this would be [1] [10]

$$0 = \lambda fx.x$$
$$1 = \lambda fx.f\,x$$
$$2 = \lambda fx.f(f\,x)$$
$$\ldots$$

1.7.4 Arithmetic operations

Now that we have defined the natural numbers in λ-notation, it is of course interesting to perform arithmetic on these numbers.

First we'll define the *successor* function, that maps a number n to its successor $n+1$. We define it to be

$$\mathbf{succ} := \lambda nfx.f(n\,f\,x).$$

This function takes the specified number, and encapsulates it another "function call", thus incrementing it by one. [10]

Next we'll define the *addition*. Just like with the successor function, we encapsulate the existing number by a function. But instead of apply f once to n, we apply another numeral m to n. The exact definition is the following

$$\mathbf{add} := \lambda mnfx.m\,f(n\,f\,x).$$

What this does is use f as the first parameter of the numeral m, the second parameter is the numeral n, so that the total amount of nested function calls amounts up to $m+n$. [10]

The *multiplication* is also possible to define using the Church numerals. What we do here, is we replace every function call in m with the function calls in n, thus multiplying the function calls. Thus instead of using n as the argument x of m like we did with addition, we use it as argument f of M. We can represent this in the λ-calculus as

$$\mathbf{mult} := \lambda mnf.m(n\,f). \quad [10]$$

One can even go as far as defining exponentiation in lambda calculus. It is also the simplest of the arithmetic formulas:

$$\mathbf{exp} := \lambda mn.m\,n \quad [13]$$

Not only can we increase Church numerals, we can also decrease them. Just like with the successor function, we'll first try and decrease the numeral by one. The process of decreasing Church numerals is somewhat more complicated, since applying a function once more is simple, but applying it once less is not. We will also define the 0 numeral to be its own predecessor i.e. $\mathbf{pred}\,0 = 0$, or in other words $\mathbf{pred}\,n = n \dot{-} 1$. With $\dot{-}$ being the *saturated subtraction*. [10]

We will define a function Φ as a helper function:

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

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

By consecutively applying this function we can go from $(0, 0)$ to $(n, n \dot{-} 1)$, thus allowing us to find the predecessor. The predecessor function can now be defined as

$$\mathbf{pred} := \lambda n.\mathbf{second}\,(n\,\Phi\lambda z.z\,0\,0) \quad [10]$$

What this basically does is apply the Φ function n times to itself, starting with $(0, 0)$ and then taking the second argument, which is de predecessor value.

The subtraction function can now be easily defined as

$$\mathbf{sub} := \lambda mn.n\,\mathbf{pred}\,m,$$

which just applies **pred** n times to m. [10]

To finalise our exploration of the arithmetic, we will define some comparison functions.

The simplest comparison function is the **iszero** function that determines whether a number is equal to zero. $\mathbf{iszero} := \lambda n.n\,(\lambda x.\mathbf{false})\,\mathbf{true}$.

The operator $\leq$ can be rewritten as $m - n \leq 0 \iff m \dot{-} n = 0$. The last bit is due to our definition of subtraction. The $\geq$ operator can be defined by switching the arguments of $\leq$.

The $<$ and $>$ operators can be defined by negating $\geq$ and $\leq$, respectively.

And lastly by combining $\leq$ and $\geq$ with a boolean *and*, we can define the $=$ operator.

The following list defines these comparison operators by using the lambda syntax:

$$\begin{aligned}
\leq &:= \lambda mn.\mathbf{iszero}(\dot{-}\,m\,n)\\
\geq &:= \lambda mn.\mathbf{iszero}(\dot{-}\,n\,m)\\
> &:= \lambda mn.\mathbf{not}(\leq\, m\,n)\\
> &:= \lambda mn.\mathbf{not}(\geq\, m\,n)\\
= &:= \lambda mn.\mathbf{and}\,(\leq\, m\,n)\,(\geq\, m\,n)
\end{aligned}$$

1.8 Recursion

Earlier, we noted that the lambda calculus only has anonymous functions. This makes recursion somewhat more difficult since constructs like $f(x) = f(x')$ are not possible. In order to make recursion possible we let functions call themselves by passing them as a parameter in a lambda abstraction. [10]

We will first introduce the *Fixed-Point Theorem*, which makes it possible to find fixed points in the λ-calculus. [4]

1. $\forall F \exists X : F\, X = X$,
2. There is a fixed point combinator

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

so that

$$\forall F : F(\mathbf{Y}\, F) = \mathbf{Y}\, F.$$

Note that other definitions of the fixpoint combinator exist. [12].

We can then use the $\mathbf{Y}$ combinator to define recursive functions as follows. Let's take $\lambda f x.e$ to be our function, with x and f being free in e. We can then represent the recursive function that satisfies $f\, x = e$ by

$$\mathbf{Y}\lambda f x.e. \quad [11]$$

This will then infinity reapply the function to itself. [7] For example

$$\mathbf{Y}\, f \overset{\beta}{\twoheadrightarrow} f(\mathbf{Y}\, f) \overset{\beta}{\twoheadrightarrow} f(f(\mathbf{Y}\, f)) \overset{\beta}{\twoheadrightarrow} \cdots$$

One should note that while this infinite recursion mechanism may not yet look useful, there exists λ-expressions so that conditional termination of the recursion is possible. [11]

As an example, we will define the factorial function in a recursive manner: [11]

$$\mathbf{Y}\, (\lambda f x.(\mathbf{iszero}\, x)\, 1\, (\mathbf{mult}\, x\, (\mathbf{pred}\, x)\, f))$$

this function recursive applies the function to itself. In each function call a check is performed on x to determine whether x is equal to zero in order to terminate the recursion.

1.9 Normalisation, termination, and Turing completeness

If a reduction strategy leads to a normal form, it is said to be *normalising*. If for a given term M all reduction strategies are normalising, the term is said to be *strongly* normalising. [4] If for a given term no reduction strategies are normalising, the term is said to be *non terminating*. [10].

If a term M has a normal form, then iterated contraction of the leftmost redex, will lead to that normal form. As such, the *leftmost reduction strategy* is normalising. [4] The leftmost reduction strategy is also called the *normal-order* reduction strategy. [12]

A few examples:

$$(\lambda x.xx)(\lambda x.xx) \xrightarrow{\beta} (\lambda x.xx)(\lambda x.xx) \xrightarrow{\beta} \cdots \tag{1}$$

$$\underline{(\lambda ab.b)}\,\underline{((\lambda x.xx)(\lambda x.xx))}(\lambda x.x) \xrightarrow{\beta} (\lambda b.b)\,(\lambda x.x) \xrightarrow{\beta} \lambda x.x \tag{2}$$

$$(\lambda ab.b)(\underline{(\lambda x.xx)}\,\underline{(\lambda x.xx)})(\lambda x.x) \xrightarrow{\beta} (\lambda ab.b)(\underline{(\lambda x.xx)}\,\underline{(\lambda x.xx)})(\lambda x.x) \xrightarrow{\beta} \cdots \tag{3}$$

$$\mathbf{I}\,\mathbf{I}\,x \xrightarrow{\beta} \mathbf{I}\,x \xrightarrow{\beta} x \tag{4}$$

(1) has no normalising reduction strategies, the term is thus non-terminating. The reduction strategy in (2) is normalising, but since the reduction strategy in (3) does not terminate, the term is not strongly normalising. And in (4) all reduction strategies terminate, it is said to be strongly normalising. [10]

Now that we have a notion of termination and non-termination, it can be interesting to address the link with the Halting Problem.

In the theory concerning Turing machines, there are Turings machines that do not halt i.e. they keep running indefinitely. Since the λ-calculus is Turing-complete, there have to be λ-expressions whose evaluation does not terminate. Indeed there are, the so called non-terminating expressions we discussed in the previous paragraphs. [10]

It is also true that determining whether a lambda expression has normal form (i.e. whether or not it terminates) is undecidable, since this problem is equivalent to determining whether a Turing machine would halt, which is also undecidable. [10]

2 Simply Typed λ-calculus

Up until now we have only considered *type-free* λ-expressions. since every expression may be applied to every other expression. We will now explore the *typed* λ-calculus, in which we apply restrictions on which expressions can be applied to other expressions. More specifically, we will explore the *simply typed* λ-calculus. These restrictions will make it easier to avoid inconsistencies and programming errors. [12].

2.1 Introduction into typing

Let $M \in \Lambda$ be a term and t be a type assigned to M, then we say "M has type t", "M in t", "$M \in t$", or "$M : t$". [5]. For example when one wants to specify the type of an argument in an abstraction, one can write $\lambda x : t.M$. [11] However, there are also implicit typing systems, in which the type of a term is not explicitly noted. [12]

In order to construct the set of types T, we first have to define a set P of *primitive types*. For example $P = \{\texttt{int}, \texttt{bool}, \texttt{float}\}$. We then construct T as follows: [11]

- $\forall t \in P : t \in T$,
- $t_1, t_2 \in T : t_1 * t_2 \in T$ (pairs),
- $t_1, t_2 \in T : t_1 \to t_2 \in T$ (functions)

Note: using the pairs, we can define functions with multiple parameters: $\texttt{int}*\texttt{int} \to \texttt{int}$. After currying this becomes $\texttt{int} \to \texttt{int} \to \texttt{int} \equiv \texttt{int} \to (\texttt{int} \to \texttt{int})$. One can even define the type of the function $\texttt{curry}_{\texttt{int}}$ as

$$(\texttt{int} * \texttt{int} \to \texttt{int}) \to (\texttt{int} \to (\texttt{int} \to \texttt{int})). \quad [11]$$

2.2 Overview of typing systems

Typing systems can be classified according to multiple criteria. For example, one can divide typing systems up into those that are based on Curry typing, and those that are based on Church typing. [4] We can also classify them based on whether they are *monomorphic* or *polymorphic*, whether they are *static* or *dynamic*, or a combination of both. [12]

It is also possible to make the distinction between typing using *explicit* typed terms in which the type is written next to the variable, and *implicit* typed terms where the type is not written next to the variable. In such a system, one has to infer the type of an expression using inference techniques. [12]

2.3 Contexts and inference

In order to reason about types, we need to keep track of the assigned types. This is done by using a *context*, which is a sequence of pairs combining variable names and their corresponding types. A context is noted Γ and the empty context is noted as $\emptyset$. Adding one variable with its type to the context can be written as $\Gamma, x : t$. [11] To note that a certain type assignments follows from a context one writes that "Γ yields M in t", or that $\Gamma \vdash M : t$. [5]. We can also view Γ as a finite function that takes a λ-term as its argument and returns the type of the λ-term.

We can write typing rules using the notations already used for inference rules like the one of following form:

$$\frac{P_1 \quad P_2 \cdots P_n}{Q},$$

in which $P_1, P_2, \cdots, P_n$ are assumed to be true, and Q is then deduced from $P_1, P_2, \cdots, P_n$. [11]

Using this notation different kinds of statements about types can be made. We can, for example, define the type of an application:

$$\frac{\Gamma \vdash M : t \to t' \quad \Gamma \vdash N : t}{\Gamma \vdash M\,N : t'}$$

What is said here is that if we can derive out of context Γ that M is a function of type $t \to t'$, and that N is a term of type t, then we can conclude that the term $M\,N$ has type t'. [11]

Using this notation we can define the typing of the basic constructs of the lambda calculus: variables, abstraction and abstraction. [11] [12]

$$\frac{\Gamma(x) \text{ is defined}}{\Gamma \vdash x : \Gamma(x)} \qquad \text{(variables)}$$

$$\frac{\Gamma, x : t \vdash M : t'}{\Gamma \vdash \lambda x : t.M : t \to t'} \qquad \text{(abstraction)}$$

$$\frac{\Gamma \vdash M : t \to t' \quad \Gamma \vdash N : t}{\Gamma \vdash M\,N : t'} \qquad \text{(application)}$$

A term M is said to be *type-correct* w.r.t. Γ if we can infer that there is a type t so that $\Gamma \vdash M : t$. The type of type-correct term is uniquely determined with respect to a context Γ. Each subterm of a type-correct term is again type-correct. [12]

This makes it possible to deduce that not every term has a type. For example self-application, noted as $\Gamma \vdash \lambda x : t.(x\,x) : t'$, is untypeable. [12]

2.4 Typed reduction and termination

In order to evaluate and manipulate the simply typed λ-calculus, we can use the same operations we use in untyped λ-calculus such as α-conversion, β-reduction, etc. [11]

An important property of the β-reduction is the so called *subject-reduction theorem* that states that the type of an expression is preserved under β-reduction. [5] [12] Suppose $M \overset{\beta}{\twoheadrightarrow} M'$ then

$$\Gamma \vdash M : t \implies \Gamma \vdash M' : t. \quad [5]$$

One can also see that the α-conversion preserves types, since it only renames bound variables, and that the η-reduction is also type-preserving since it only removes encapsulation.

It should be noted that the opposite of the β-reduction, namely the β-expansion does not preserve typing, since it is often possible to derive type-correct terms from untypable terms. [5] [8] [12]

Another interesting property of the β-reduction when applied on type-correct terms, is that it is guaranteed to terminate. [12] We can thus say that all typable expressions in simply typed λ-calculus are strongly normalising. [11] β-equivalence is also decidable for type-correct terms. [12]

There exists a proof of the termination theorem, but for reasons of brevity it will not be discussed here. Intiutively we can say that due to typing, self-application is forbidden in the simply typed lambda calculus, thus making recursion impossible. [11] [12] An intuitive reason why self-application is forbidden, is that the type of a function is strictly larger than the type of its argument (See: the definition of the type an abstraction), which has as a consequence that a function cannot be used as its own argument. [11]

While guaranteed termination may seem like a good thing, which it partly is, it also causes us to lose the possibility to define recursive functions. [11] It is, however, possible to reintroduce the **Y**-combinator as a constant with a "hardcoded" reduction rule

$$\mathbf{Y}\, f \overset{\beta}{\twoheadrightarrow} f(\mathbf{Y}\, f),$$

but this also reintroduces non-termination. This technique is used in a number of functional programming languages. [11] [12]

A downside of simply typed λ-calculus is that its not Turing-complete. [12] Again due to brevity reasons we will not give a formal proof here. We will however intuitively reason about what causes this Turing-incompleteness.

A first reason is the absence of recursion, which also prevents iterative algorithms from being implemented. However, if we reintroduce the **Y**-combinator as a constant, all computable functions can be represented [12]

Another reason is that all type-correct expressions are guaranteed to be terminating. Since a terminating lambda term is equivalent to a halting turing machine, this means that non-halting turing machines cannot be represented. [12]

3 References

[1] Alonzo Church. *An Unsolvable Problem of Elementary Number Theory.* 1936.

[2] David C. Luckham, Nils J. Nilsson. *Extracting Information from Resolution Proof Trees* 1971.

[3] Thomas W. Hungerford. *Algebra.* 1974.

[4] H.P. Barendregt. *Lambda Calculi with Types.* 1991.

[5] H.P. Barendregt, Erik Barendsen. *Introduction to Lambda Calculus.* 1998.

[6] Terese. *Term Rewriting Systems* 2003.

[7] Georg P. Loczewski. *The Lambda Calculus: A Brief Introduction.* 2005.

[8] Erik Barendsen. *Introduction to Type Theory..* 2005.

[9] Felice Cardone, J. Roger Hindley. *History of Lambda-calculus and Combinatory Logic*. 2006.

[10] Manuel Eberl. *The untyped Lambda Calculus*. 2011.

[11] Yves Bertot. *Lambda-calculus and types*. 2015.

[12] Tobias Nipkow. *Lambda Calculus*. 2018.

[13] Dr. David Stotts. *Lambda calculus*. URL: `http://www.cs.unc.edu/ stotts/723/Lambda/church.html`. Accessed on 2019-04-13.