Abstract

The untyped lambda-calculus is a concept to describe computability. It has a mathematical notation that allows to formalize the evaluation of functions. Functions in the untyped lambda-calculus can also be translated to a program that can run on a machine.

The untyped lambda-calculus is a topic that can be difficult to understand. There is a lot of literature available on this topic. However there is a large gap between very high-level information sources that are hard to understand and the simple articles on wikipedia that often lack sophisticated sources.

This document fills this gap by giving a broad overview over the topic of the untyped lambda-calculus and discussing the points that are relevant for software engineers in detail.

At the end of the article, a basic implementation of the untyped lambda-calculus in C# will be shown.
## Contents

1 Introduction .................................................. 3
   1.1 The $\lambda$-calculus ................................... 3
   1.2 $\lambda$-calculus usages .................................. 3
   1.3 Models of computation .................................... 3
   1.4 The Church-Turing thesis ................................. 4

2 Basics of the $\lambda$-calculus ................................. 5
   2.1 Syntax ..................................................... 6
   2.2 Operational semantics .................................... 8
   2.3 Nameless representation of terms ....................... 9

3 Programming in the lambda calculus ......................... 10
   3.1 Untyped expressions ..................................... 10
   3.2 Evaluation ............................................... 11
   3.3 Arithmetic expressions .................................. 15
   3.4 Stuck terms ............................................. 15

4 Enrichment of the lambda calculus ......................... 15
   4.1 Multiple arguments ..................................... 16
   4.2 Church Booleans ........................................ 16
   4.3 Church Numerals ........................................ 17
   4.4 Recursion ............................................... 17

5 $\lambda$-calculus via C# ..................................... 17
   5.1 C# concepts ............................................ 17
   5.2 Church Booleans ........................................ 19

6 Conclusion ................................................... 21
1 Introduction

1.1 The $\lambda$-calculus

The lambda-calculus is a formal language that allows to study functions. It is described by the definition of functions and bound parameters. It was introduced in 1930 by Alonzo Church and Stephen Cole Kleene. [Jun04]

In the lambda-calculus, everything is a function. The following example explains what that means:

Consider the number 3. It can also be written as $0 + 1 + 1 + 1$. If we now define a function $\text{succ}(x) \rightarrow x + 1$ we can also write 3 as $\text{succ}(\text{succ}(\text{succ}(0)))$. This approach allows us to define an arbitrary number of functions which can take other functions as parameters and also yield functions as result.

This alternative approach can be seen as a new interpretation of mathematics. In contrary to the axiomatic approach which has sets as basic objects, the lambda-calculus has universal functions as basic objects.[Chu32]

How this approach can be useful will be explained in this document.

1.2 $\lambda$-calculus usages

The lambda-calculus has two key roles:

- It is a simple mathematical foundation of sequential, functional, higher-order computational behaviour.
- It is a representation of proofs in constructive logic.

Each role for itself is not very exciting. Its the combination of the two that makes the concept interesting. It allows to formulate a computation in a mathematical way by formulating lambda-terms. These terms can be translated into a program. Now if this program compiles correctly, it can be regarded as a proof that the mathematical formulation was correct as well. It also works the other way and makes it possible to verify programs in a mathematical way. This is also known as the Curry-Howard correspondence[Gri15]. It represents a dual view of the lambda-calculus, as mathematical proof or as programming language (sequential, functional, higher-order languages).

As for the theory domain, the lambda-calculus is used in the areas of mathematics, philosophy[Cud07] and linguistics[Moo88].

For a more practical domain, the lambda-calculus played a major role in the development of the theory of programming languages[mod]. Furthermore, functional programming languages implement the lambda-calculus[Mic11]. For example Haskell has a syntax that is almost equivalent to the lambda-calculus syntax.

These concepts can be formalized to the general model of computability which is described in the next section.

1.3 Models of computation

Computation is often specified as how the evaluation of a function or program can progress. This is expressed as the description of some kind of conceptual automation[mod].
Although it is mathematically unprovable, the thesis exists that there is a reasonable intuitive definition of "computable"[Squ]. This thesis is equivalent to this list of provably equivalent formal models of computation:

- Turing machines
- Lambda Calculus
- Post Formal Systems
- Partial Recursive Functions
- Unrestricted Grammars
- Recursively Enumerable Languages

Naturally speaking, these models describe what is computable by a computer program written in any reasonable programming language. This thesis is also known as the Church-Turing thesis described in the next section 1.4.

1.4 The Church-Turing thesis

The Church-Turing thesis is a hypothesis about the nature of computable functions. It states that a function on natural numbers is computable if it is computable by a turing machine.

The notion of the Church-Turing thesis is, to provide an effective or mechanical method in logic and mathematics. Effective and its synonym mechanical are terms to describe a function M.[Cop02]

A function M is called effective or mechanical just in case all of the following statements are true:

- M is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);
- M will, if carried out without error, produce the desired result in a finite number of steps;
- M can (in practice or in principle) be carried out by a human being unaided by any machinery, only with paper and pencil;
- M demands no insight or ingenuity on the part of the human being carrying it out.

This set of statements can also be interpreted in relation to the lambda-calculus:

- A lambda term is reduced to its normal form in a finite number of reductions
- In the untyped lambda-calculus a well formed lambda-term has exactly one derivation tree without branches and therefore is guaranteed that the terms final derivation is an axiom
• Every lambda term has a finite length so it can be reduced with pen and paper within a finite amount of space in a finite amount of time.

• A lambda term on paper can be compiled into a program that can be executed on a computer

This leads to the assumption that both Turing Machines and the lambda-calculus can make the statement: Is a function calculable by a machine? The rest of this document will now examine the lambda-calculus in detail and show that the statements mentioned above hold.

2 Basics of the $\lambda$-calculus

The key feature of the lambda-calculus is functional abstraction. Abstracted functions are exactly what is needed to drive efficient and deterministic computation both in theory, proofs, simulations and programming languages. This was explained in section 1.2 and section 1.4.

Abstraction allows us to write a function that executes the code generically instead of writing the same code over and over again. Each time the function is computed, it is instantiated with an individual set of one or more named parameters that alters the outcome of the computation. Upon instantiation, a value for each parameter is needed.

For example consider the difference of two factorials:

$$(5\cdot4\cdot3\cdot2\cdot1) - (3\cdot2\cdot1)$$

This term would be much easier to write as:

$$\text{factorial}(5) - \text{factorial}(3)$$

It is quite natural to implement the factorial as a recursive function. In pseudo code one could write:

$$\text{factorial}(n) = \text{if } n=0 \text{ then } 1 \text{ else } n \cdot \text{factorial}(n-1)$$

In the untyped lambda-calculus we can write “$\lambda n \ldots$” instead which means “the function that for $n$, yields…”:

$$\text{factorial} = \lambda n. \text{ if } n=0 \text{ then } 1 \text{ else } n \cdot \text{factorial}(n-1)$$

In the lambda-calculus everything is a function. A function can only have other functions as arguments and it can only yield another function as result, so to speak. How this is achieved is explained in this section. The information in this section is derived from the book Types and Programming Languages from Benjamin C. Pierce.[Pie02]
2.1 Syntax

The syntax of the lambda-calculus is surprisingly simple. It consists of only three kinds of terms. This can be summarized by the following grammar:

\[
\begin{align*}
(1) \quad t & \ ::= \ x & \quad \text{(variable)} \\
(2) \quad t & \ ::= \ \lambda x. \ t & \quad \text{(abstraction)} \\
(3) \quad t & \ ::= \ t \ t & \quad \text{(application)}
\end{align*}
\]

The next section will briefly discuss the three different types of terms.

2.1.1 The grammar in detail

To describe the grammar, we will look at each rule individually and examine how it can help to abstract problems from the real world.

Variable A variable \(x\) (and \(y,z\)) by itself is a term. It represents a function which cannot be further evaluated. A variable is a term in normal form as explained in section 3.2.1. These variables are also called object language variables. This can be thought of as if this variables have a certain meaning in the context of the language. For example \texttt{true} can be such a variable because it might give an exact meaning to the outcome of a calculation which a person can understand.

\(t\) (and \(s,u\)) are meta-variables. They can stand for an arbitrary term. These terms have to be lambda-terms. This basically means that terms can be further reduced (evaluated). This is explained in section 3.2.

Application In the lambda-calculus, function application is the act of applying a function to an argument, which in the lambda calculus is another function. Constant arguments like booleans or number literals are considered as terms in normal form, since they can not be computed any further. Nevertheless, they are also functions as we will see in section 3.1.

In mathematics, functions are usually written \(f(x)\). In the lambda calculus the parentheses are usually omitted. We can simply write \(f \, x\).

Since the result and parameters in the lambda-calculus are also functions this leads to the question in what order they are executed. According to Pierce [Pie02] application associates to the \textbf{LEFT}. For example \(s \, t \, u\) stands for \((s \, t) \, u\). That means that \(u\) is applied to the result function of the \(s \, t\) applications.

Abstraction Abstraction is a technique to separate complexity into a hierarchy. In the syntax of the lambda-calculus abstraction is introduced in rule (2) where \(t\) appears on the right side of the definition. Since \(t\) can be of form (2) as well, it is possible to formally describe a computation over multiple layers.

\[(\lambda x. \ x)(\lambda y. \ ((\lambda z. \ z) \ t) \ u) \ s\]

Like with the application, the parentheses for abstraction are usually omitted. According to Pierce [Pie02], abstraction associates to the \textbf{RIGHT}. For example \(\lambda x. \lambda y. \ x \ y \ x\) stands for \(\lambda x. (\lambda y. ((xy)x))\).
2.1.2 Scope of variables

Variables can be bound or free in the context of an abstraction. If we consider the abstraction \( \lambda x.t \) then \( \lambda x \) is a binder whose scope is \( t \).

- \( x \) is bound when it occurs in the body of \( t \)
- \( x \) is free when it is not bound by an enclosing abstraction (binder) on \( x \)

The following examples show the difference between bound and free variables:

\[
\begin{align*}
x & \ y \quad (\text{occurrences of } x \text{ are free}) \\
\lambda y. \ x \ y & \quad (\text{occurrences of } x \text{ are free}) \\
\lambda x. \ x & \quad (\text{occurrences of } x \text{ are bound}) \\
\lambda z. \lambda x. \lambda y. \ (y \ z) & \quad (\text{occurrences of } x \text{ are bound}) \\
(\lambda x. \ x) \ x & \quad (\text{first occurrence of } x \text{ is bound second is free})
\end{align*}
\]

2.1.3 Abstract and concrete syntax

Because of the Curry-Howard correspondence [Gri15] it is at least necessary to have two representations of the same program. One that is machine readable and one that is easy to write for a programmer.

What a programmer writes is usually source code. While a programmer can read source code and understand it, a machine has to do additional steps to be able to execute a program. This leads to concrete and abstract syntax which are two different representations of the same calculation.

**concrete syntax:** String of characters the programmer writes directly

**abstract syntax:** Simpler internal representation as labeled trees (ASTs)

**Transformation** Abstract syntax is a natural fit for compilers. But because concrete syntax is what the programmer writes, there must be a way to convert between the two.

**From concrete to abstract syntax** The conversion from source code to an AST usually includes a number of steps that are specific for a concrete implementation of the language. But almost every time they contain a lexer and a parser.

The lexer converts the source file into a stream of tokens and also gathers as much information about them as possible (keyword, constant, punctuation, etc.). Additionally it also takes care of dumping white spaces and comments from the code since they have no relevancy for the execution of the computation.

The parser then converts the sequence of tokens into an AST which can be processed by a machine. During this process operator precedence and associativity is taken into account. This leads to less parentheses to be written by the programmer.
From abstract to concrete syntax  It is also possible to convert an AST back
to concrete syntax. This can be accomplished with a print visitor that visits the
whole AST and prints every node in order. Although white-spaces and comments
can not be recovered this way.

2.2 Operational semantics

The most basic concept of computation can be seen in the application of function
to arguments (which themselves are functions). All other primitives that are com-
mon in programming languages like numbers, arithmetic operations, conditionals,
records, loops, sequencing, I/O, etc. are built on top of this concept. This will be
examined in detail in section 3.

2.2.1 Computation

The computation of terms which are reducible can be subdivided into steps. Each
step of computation consists of rewriting an application whose left hand side is an
abstraction.

Rewriting means substituting the right hand component of the application
for the bound variable in the abstraction body. This is shown in the following
example:

\[(\lambda x. t_{12}) t_2 \rightarrow [x \rightarrow t_2]t_{12}\]

\([x \rightarrow t_2]t_{12}\) means that all free occurrences of \(x\) in \(t_{12}\) have to be replaced by \(t_2\).

Additional examples:

\[(\lambda x. x) y\] evaluates to \(y\) (identity function)
\[(\lambda x. x)(\lambda z. x)\) evaluates to \((\lambda x. x)(\lambda z. x)\)

A term in the form \((\lambda x. t_{12}) t_2\) is called a redex (“reducible expression”). A redex
is a term on which a step of evaluation can be performed. The rewrite operation
described here is also called \textbf{beta-reduction}[Pie02].

It would be interesting to know why the phrase beta-reduction was chosen to
describe this kind of process. Sadly I could not find any information, who invented
this phrase.

2.2.2 Evaluation strategies

There are several evaluation strategies how a redex can be reduced. To show these
different approaches the sample lambda-term \((\lambda x. x)((\lambda x. x)(\lambda z. (\lambda x. x) z))\) is
simplified to: \textit{id(id(id z))}. The id function in this example is simply a shorter
form of writing \(\lambda x. x\) which is the identity function that just returns its argument.
**Full beta-reduction**  At each step of computation we can choose a redex anywhere inside the term and reduce it. There is no particular rule, which redex is chosen for reduction at each step of evaluation. In the following example we choose to always reduce the innermost redex first:

\[
\text{id}(\text{id}(\lambda z. \text{id} z)) \rightarrow \text{id}(\lambda z. \text{id} z) \rightarrow \lambda z. \text{id} z \rightarrow \lambda z. z
\]

**Normal order strategy**  The leftmost, outermost redex is reduced first:

\[
\text{id}(\lambda z. \text{id} z) \rightarrow \lambda z. \text{id} z \rightarrow \lambda z. \text{id} z \rightarrow \lambda z. z
\]

**Call by name**  This strategy is more restrictive because it does not allow reductions inside abstractions. When this criteria is not fulfilled anymore, the resulting term is regarded as normal form. Example:

\[
\text{id}(\lambda z. \text{id} z) \rightarrow \lambda z. \text{id} z \rightarrow \lambda z. \text{id} z \rightarrow \lambda z. z
\]

**Call by value**  This strategy appears in most languages. The outermost redex is reduced first and it will only be reduced if its right hand side has already been reduced to a value before.

\[
\text{id}(\lambda z. \text{id} z) \rightarrow \lambda z. \text{id} z \rightarrow \lambda z. \text{id} z
\]

In the pure untyped lambda-calculus there are only values that are lambda-terms in normal form. Richer calculi will allow additional values like: numeric and boolean constants, strings, etc.

The *call by value* strategy is very strict because the arguments of functions are evaluated regardless if they are used in the body or not. In contrast, non-strict (or lazy) strategies, like the strategies above, evaluate only the arguments that are actually used.

In many implementations however, this eager evaluation strategy is often realized with the lazy evaluation where arguments are only evaluated if they actually appear in the function body. This is mainly done to avoid repeating evaluations of the same argument.

One reason why *call by value* is so popular is that it is easy to enrich it with features like exceptions and references. This can be read in Pierce chapter 14 and 13 [Pie02].

### 2.3 Nameless representation of terms

Another problem that is worth mentioning is that of variable name clashes. If a free variable in a term happens to be the same as a variable under substitution it would give the term a different meaning. For example in the body of the abstraction of the term \((\lambda x. x + y)(y)\), the free variable \(y\) would have nothing to do with the argument \((y)\).

To solve these problems there are many approaches, and to mention them all is out of the scope of this document. However a simple approach is just to rename
every variable consequently to a unique name (more exactly: to a number). This guarantees that there are no name clashes.

Church used the term \textit{alpha-conversion} for the operation of consistently renaming a bound variable in a term\cite{Pie02}.

3 Programming in the lambda calculus

After analyzing the untyped lambda-calculus in a formal way, it is now time to get our hands dirty and examine the specifics of the untyped lambda-calculus in detail.

First we define a basic grammar that can be expressed in the untyped lambda-calculus. As this section goes on, we will improve the language with extensions that are common for programming languages. The theoretical part of this section will be written according to Pierce \cite{Pie02}.

3.1 Untyped expressions

Since the untyped lambda-calculus does not include types, we call the language, that we discuss here, \textbf{untyped expressions}. This is a very vague definition, and there are several ways to formalize untyped expressions. The most intuitive way is to define a grammar.

3.1.1 Grammar

\begin{align*}
t & ::= \text{true} \mid \text{false} \quad \text{(constant true/false)} \\
t & ::= \text{if } t \text{ then } t \text{ else } t \quad \text{(conditional)} \\
t & ::= 0 \quad \text{(constant zero)} \\
t & ::= \text{succ} \ t \quad \text{(successor)} \\
t & ::= \text{pred} \ t \quad \text{(predecessor)} \\
t & ::= \text{iszero} \ t \quad \text{(zero test)}
\end{align*}

This simple grammar will be the first thing we use to describe untyped systems. The style of the grammar is close to the BNF which is suited for simpler parsing by machines very well. On the right hand side of the rules, \( t \) may substitute to any other term \( t \). For readability, parentheses are omitted.

3.1.2 Syntax

There are several different ways of defining the syntax of untyped expressions. One is the \textbf{grammar} that we saw in the last paragraph 3.1.1. The next three paragraphs will describe different approaches.

\textbf{Inductively} \hspace{1em} Untyped expressions are defined with the theory of sets. Since this is very theoretical and not machine-readable, we will not discuss this approach in more detail.
Inference rules  Inference rules are a very visual way to describe untyped expressions. With inference rules, the set of terms from the above grammar is defined as follows:

\[
\begin{align*}
\text{true} & \in T \\
\text{false} & \in T \\
0 & \in T \\
\text{succ } t_1 & \in T \\
\text{pred } t_1 & \in T \\
\text{iszero } t_1 & \in T \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \in T
\end{align*}
\]

Inference rules are read: “If we have established the statement in the premise(s) listed above the line, then we may derive the conclusion below the line.” Rules without premises are called axioms.

This leads to the scheme: Premises including a meta-variable lead to an infinite set of rules, obtained by replacing each t by every possible term.

Concretely  Lastly, it is possible to construct untyped expressions explicitly. The idea behind this is to start with an empty set of terms and successively add every possible term in a recursive (inductive) fashion.

3.1.3 Induction on terms

It is also worth mentioning that with induction on terms some information can be gained about terms.

Size  It is possible to derive the size of a term in a recursive fashion:

\[
\begin{align*}
(1) \text{size(true)} &= \text{size(false)} = \text{size(0)} = 0 \\
(2) \text{size(succ } t_1) &= \text{size(pred } t_1) = \text{size(iszero } t_1) = \text{size(t_1)} + 1 \\
(3) \text{size(if } t_1 \text{ then } t_2 \text{ else } t_3) &= \text{size(t_1)} + \text{size(t_2)} + \text{size(t_3)} + 1
\end{align*}
\]

The size of a term is equal to the number of nodes that the term would have in an AST representation. It also represents the number of constants + the number of redexes.

Depth  It is also possible to derive the depth of the AST of a term with induction. The formulas (1) and (2) are identical to the calculation of size. Rule (3) is:

\[
\text{depth(if } t_1 \text{ then } t_2 \text{ else } t_3) = \max(\text{depth(t_1)} + \text{depth(t_2)} + \text{depth(t_3)}) + 1
\]

3.2 Evaluation

Now that we are able to form terms according to a syntax, the next step is to evaluate such a term.

For evaluation, we define a set of values that are possible final results of evaluation. If a term cannot be evaluated any further it is in normal form. If the normal form is a value, then it is the result of a calculation. If it is not a value, the computation got stuck (see section 3.4).
To be able to evaluate a term, we augment our simple sample grammar from 3.1.1 with rules for evaluation. In a first step this is shown for boolean types. Evaluation rules \((t \rightarrow t')\) mean, that \(t\) evaluate to \(t'\) in one step of evaluation, e.g. a machine in state \(t\) can at any given moment perform one step of evaluation to change its state to \(t'\).

\[
\begin{align*}
\text{if true then } t_2 \text{ else } t_3 & \rightarrow t_2 \quad \text{(E-If-True)} \\
\text{if false then } t_2 \text{ else } t_3 & \rightarrow t_3 \quad \text{(E-If-True)} \\
\end{align*}
\]

\[
\begin{align*}
t_1 & \rightarrow t'_1 \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 & \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 \quad \text{(E-If)}
\end{align*}
\]

Because these rules are inference rules it is easy to see the dynamic of computation that they are representative for. The first two rules are axioms and therefore if we have a term in the form on the left hand side of the rules, we are guaranteed to receive the term on the right.

In the third rule, the inference notation tells us that if another machine can take one step to get from \(t\) to \(t'\), our machine can take one step to get from \(\text{if } t_1 \text{ then } t_2 \text{ else } t_3\) to \(\text{if } t'_1 \text{ then } t_2 \text{ else } t_3\). This means that for one instance of an inference rule (explained in the next section), the premises of it can be regarded as axioms for the instance.

### 3.2.1 Formal definition

Pierce [Pie02] formulated evaluation of terms with a number of definitions that define if and how a term must be evaluated according to the untyped lambda-calculus. This section briefly discusses these definitions.

To evaluate a conditional term in the form described in the grammar above, its guard has to be evaluated first. So if we encounter a term that is a conditional we can make two distinctions according to what rule shall be applied:

- **Computation rule**: If the E-IfTrue or E-IfFalse rule applies, it tells us what to do when we reach the end of this terms evaluation.
- **Congruence rule**: E-If tells us that we can evaluate the guard and therefore more work has to be done to evaluate the term.

The rest of this section explains the formal steps of how the reduction of a term is done. Essentially it is a process that maps a lambda-calculus term that contains reducible expressions to one particular reducible expression that is going to be reduced next.

**Instance of a rule** For one particular inference rule, an instance of it can be obtained by consistently replacing each meta-variable with a different term. This has to be done for all meta-variables in the rules conclusion and all its premises. It is important to notice that the variables that appear in the conclusion, or one or more of the premises, have to be replaced with the SAME term per variable. To clarify this, consider the following two examples:
**E-If-True**  Rule without a premises:

\[
if \text{ true} \ then \ t_2 \ else \ t_3 \rightarrow t_2
\]

In this example we will perform the following replacements to receive one instance of the rule:

- \( t_2 \rightarrow \text{true} \)
- \( t_3 \rightarrow \text{if false then false else false} \)

The result of the replacement operation yields the result:

\[
if \text{true} \ then \text{true} \ else \ (\text{if false then false else false}) \rightarrow \text{true}
\]

**E-If**  rule with one premises:

\[
\begin{align*}
t_1 & \rightarrow t'_1 \\
if \ t_1 \ then \ t_2 \ else \ t_3 & \rightarrow \ if \ t'_1 \ then \ t_2 \ else \ t_3
\end{align*}
\]

Replacements:

- \( t_1 \rightarrow \text{if true then true else false} \)
- \( t'_1 \rightarrow \text{true} \)
- \( t_2 \rightarrow \text{true} \)
- \( t_3 \rightarrow \text{false} \)

Result:

\[
\text{if (if true then true else false)} \ then \text{true else false} \rightarrow \text{if true then true else false}
\]

**Satisfaction of a rule**  The next step is to determine if a rule can be satisfied or not. This is done by introducing the idea of a relation for each rule. Each instance of the rule has to satisfy the relation. There are two cases to consider in which the rule is satisfied:

- The conclusion of the rule is in the relation
- One of the premises is NOT in the relation
This means if one of the premises is not derivable, the whole relation is not. The above example of the E-If rule instance would satisfy the relation of the premises and therefore CAN take one step of evaluation.

Let’s make another example and consider the premises of the E-If rule:

\[
\begin{align*}
&\quad t_1 \rightarrow t'_1 \\
&\quad \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\
&\quad \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3
\end{align*}
\]

It is NOT derivable with the following replacements:

- \( t_1 \rightarrow \text{if } \text{true} \text{ then } \text{true} \text{ else } \text{false} \)
- \( t'_1 \rightarrow \text{false} \)

The premises does not hold which means that this instance of the rule is not satisfied:

\((\text{if } \text{true} \text{ then } \text{true} \text{ else } \text{false}) \not\rightarrow \text{false}\)

**One-step evaluation** If a given term satisfies the evaluation relation that is described above, it can take one more step of evaluation. The derivation of an instance of a rule can be seen as a tree. The internal nodes represent the congruence rules and its leafs represent the computation rules.

Example:

\[
\begin{align*}
s &::= \text{if } \text{true} \text{ then } \text{false} \text{ else } \text{false} \\
t &::= \text{if } s \text{ then } \text{true} \text{ else } \text{true} \\
u &::= \text{if } \text{false} \text{ then } \text{true} \text{ else } \text{false}
\end{align*}
\]

\[
\begin{align*}
&\quad \text{E-IfTrue} \\
&\quad \text{if } \text{true} \text{ then } \text{false} \text{ else } \text{false} \rightarrow \text{false} \\
&\quad \text{E-If} \\
&\quad \text{if } s \text{ then } \text{true} \text{ else } \text{true} \rightarrow \text{if } \text{false} \text{ then } \text{true} \text{ else } \text{true} \\
&\quad \text{E-If} \\
&\quad \text{if } t \text{ then } \text{false} \text{ else } \text{false} \rightarrow \text{if } u \text{ then } \text{false} \text{ else } \text{false}
\end{align*}
\]

It is important to notice that in our sample grammar all rule instances lead to a distinct derivation tree. This is because there are no rules with more than one premises which could lead to branches in the tree.

**Normal form** A term is in normal form when no evaluation rule applies to it. This also means that values are always in normal form, since they can not be evaluated further. The contradiction is also true in most cases: if \( t \) is in normal form, its a value. It only fails if a term is not computable (stuck).
Multi-step evaluation  It is deterministic, that a series of single-step evaluations result in the same computation rule.

3.3 Arithmetic expressions

It is natural for computer programs to perform calculation on numbers so it makes sense to extend the simple language that only supports booleans until now to support numbers (arithmetic expression).

Since there is an infinite number of natural numbers, we can not define them all as values. Instead, we use lambda-terms to construct natural numbers.

For this purpose a new type of variables is introduced, the numeric values. This syntactical construct has the intention of providing a method to allow numeric values as final result of evaluation. This concept also allows to define rules to prevent malformed terms like for example \( \text{succ}(\text{true}) \) which should be an error and not a value.

This introduces additional evaluation rules to support natural numbers. The following rules need to be added:

\[
\begin{align*}
\text{succ } t_1 & \rightarrow \text{succ } t'_1 & & \text{(E-Succ)} \\
pred 0 & \rightarrow 0 & & \text{(E-PredZero)} \\
pred(\text{succ } nv_1) & \rightarrow nv_1 & & \text{(E-PredSucc)} \\
t_1 & \rightarrow t'_1 & & \text{(E-Pred)} \\
iszero 0 & \rightarrow \text{true} & & \text{(E-IsZeroZero)} \\
iszero(\text{succ } nv_1) & \rightarrow \text{false} & & \text{(E-IsZeroSucc)} \\
t_1 & \rightarrow t'_1 & & \text{(E-IsZero)}
\end{align*}
\]

3.4 Stuck terms

It has already been mentioned several times in this document, that there are situations when a calculation gets stuck. Formally spoken: A closed term (term in which all variables are bound) if it is in normal form and not a value. “Stuckness” is the equivalent for run-time errors.

4 Enrichment of the lambda calculus

In order to improve the pure untyped lambda-calculus there are several concepts that can be easily implemented with the existing infrastructure to provide common tools for programmers.
4.1 Multiple arguments

By default, the untyped lambda-calculus does not support multiple arguments. Instead of adding new rules to support multiple arguments it is easier to achieve the same effect using currying of higher-order functions (functions that can take multiple steps of evaluation).

So to apply multiple arguments to a function, the first argument is applied to the function. The second argument then can be applied to the function that was yielded as a result from the first function. This process is repeated until all arguments are applied.

4.2 Church Booleans

In the lambda-calculus it is also possible to evaluate boolean conditionals. This is possible by defining lambda-terms that behave exactly like the evaluation rules mentioned in 3.2.

If we for example want to evaluate the E-If rule:

\[
\frac{t_1 \to t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3} \to \text{if } t_1' \text{ then } t_2 \text{ else } t_3
\]

We can defined a function \(\text{test}\ b\ v\ w\) that reduces to \(v\) or \(w\) depending on whether the boolean \(b\) is true or false. It is now possible to define combinators that yield the expected reduction:

\[
\begin{align*}
\text{true} &= \lambda t. \lambda f. t; & \text{ (true combinator)} \\
\text{false} &= \lambda t. \lambda f. f; & \text{ (false combinator)} \\
\text{test} &= \lambda l. \lambda m. \lambda n. l\ m\ n; & \text{ (test combinator)}
\end{align*}
\]

This example shows the reduction of an example term \(\text{test}\ \text{true}\ v\ w\) that yields \(v\) as result:

\[
\begin{align*}
\text{test}\ \text{true}\ v\ w
&= (\lambda l. \lambda m. \lambda n. l\ m\ n)\ \text{true}\ v\ w & \text{(replace test with definition)} \\
&\to (\lambda m. \lambda n. \text{true}\ m\ n)\ v\ w & \text{(reducing the leftmost redex)} \\
&\to (\lambda n. \text{true}\ v\ n)\ w & \text{(reducing the leftmost redex)} \\
&\to \text{true}\ v\ w & \text{(replace true with definition)} \\
&\to (\lambda f. t)\ v\ w & \text{(reducing the leftmost redex)} \\
&\to (\lambda f. v)\ w & \text{(reducing the leftmost redex)} \\
&\to v & \text{(result term)}
\end{align*}
\]

In the same fashion it is also possible to define boolean operators like AND, OR, etc.
4.3 Church Numerals

Similar to Church Booleans it is also possible to define lambda functions to operate on natural numbers.

4.4 Recursion

Recursion causes kind of a break of the lambda-calculus. A recursive term can never evaluate to normal form and therefore a term would not be guaranteed to evaluate. This is why in the pure lambda-calculus there is no recursion.

Although I will not fully explain how recursion can be implemented regardless, I want to mention the \( \omega \) combinator, which is the key idea to implement recursion. This divergent combinator will always return itself:

\[
\omega = (\lambda x . x x)(\lambda x . x x)
\]

Now in order to fully implement recursion also the \( fix \) combinator is needed to define the break condition of a recursive function. With these two combinators it is possible to design recursion.

5 \( \lambda \)-calculus via C#

The Blog of Yan Dixin [Dix15] contains a sophisticated section about the lambda-calculus via C#. This section will discuss some of his examples that represent the concepts that have already been discussed above in this document.

5.1 C# concepts

In his article, Dixin deeply examines lambda expressions and the lambda calculus - how it comes, what it does, and why it matters. And - all of that is accomplished by using only functions and anonymous functions. He uses C# functions to demonstrate how lambda expressions model a computation.

5.1.1 C# closures

The base builds a simple concept in C# called closure. It is actually a general concept that is also used in other programming languages. “In computer science, a closure is a first-class function with free variables that are bound in the lexical environment.” [Eth10] In Listing 1 the variable \( \text{foo} \) is a free variable, only bound by the environment. The fact that \( \text{myFunc} \) accesses this variable, is called a closure since it is not local in \( \text{myFunc} \).

```csharp
1 var foo = "this is over 9000";
2 Func<string, string> myFunc = delegate (string arg) {
3     return arg + foo;
4 }
```

Listing 1: Closures in C#
5.1.2 Currying and partial application

To write code in C# in a lambda style fashion, several language features can be used. The most important concepts are shown in Listing 2.

```csharp
private static void LambdaBasics()
{
    // Simple function
    Func<int, int, int> add = (x, y) => x + y;
    int result = add(1, 2);

    // Lambda style - anonymous function without name
    result = new Func<int, int, int>((x, y) => x + y)(1, 2);

    // Apply multiple arguments with currying
    Func<int, Func<int, int>> curriedAdd = x => new Func<int, int>(y => x + y);

    // The return type function declaration can be inferred by
    // the compiler
    curriedAdd = x => y => x + y;

    // Applying add to the first argument yields a function
    // as result
    Func<int, int> add1 = curriedAdd(1);

    // Now the second argument can be applied to add1
    result = add1(2);
}
```

Listing 2: Currying and partial application

Line 4 in Listing 2 shows the declaration of a simple function which is evaluated on line 5.

The declaration of a function can also be omitted when an anonymous function is used (line 8). To support longer lambda terms, functions can also be curried. The declaration of this can be seen on line 11 and 14. The partial application of the function can be seen on line 17 and 19.

It is important to notice, that the yielded result of a function in this case is another function, which is passed to the next function as a closure.

5.1.3 => associativity

The C# lambda operator is right-associative. Because of this fact, we can omit the brackets in the examples to improve readability. \( x \Rightarrow y \Rightarrow x + y \) for example is identical to \( x \Rightarrow (y \Rightarrow x + y) \).

This is important to understand the evaluation order of lambda-expressions in C#. With curried application, always the leftmost redex is reduced first. More information on evaluation strategies can be found in section 2.2.2.

Now to clarify this we will look at the lambda term \( \lambda x.\lambda y.x + y \). In C# it can be written \( x \Rightarrow y \Rightarrow x + y \). We will evaluate this lambda term by applying two arguments, to both show the lambda derivation and the equivalent curried application in C#.
In this example for simplification, we assume that the constants 1 and 2 and the symbol + are values (object language variables).

\[
\begin{align*}
(\lambda x.\lambda y. x + y) (1) & \quad \text{(reducing the leftmost redex)} \\
\rightarrow (\lambda y. 1 + y) (2) & \quad \text{(reducing the leftmost redex)} \\
\rightarrow 1 + 2 & \quad \text{(final result)}
\end{align*}
\]

And now the equivalent in C#:

```csharp
1 Func<int, Func<int, int>> add = x => y => x + y;
2
3 // add1: y => 1 + y;
4 Func<int, int> add1 = add(1);
5
6 // result: 1 + 2
7 int result = add1(2);
```

Listing 3: Example reduction

### 5.2 Church Booleans

Encoding Church booleans in C# is surprisingly easy and serves as a good example of how to implement something useful with the lambda-calculus in C#.

#### 5.2.1 Encoding Church Booleans

Listing 4 shows how the Church Booleans from section 4.2 can be implemented in C#.

```csharp
1 private static void ChurchBooleans()
2 {
3     // True := \lambda t. \lambda f. t
4     Func<object, Func<object, object>> True =
5         (object @true) => @false => @true;
6
7     // False := \lambda t. \lambda f. f
8     Func<object, Func<object, object>> False =
9         (object @true) => @false => @false;
10
11     // True Test
12     object result = True(1)("2"); // result = 1
13     result = True("a")(null); // result = "a"
14     object @object = new object();
15     result = True(@object)(null); // result = @object
16
17     // False Test
18     result = False(1)("2"); // result = "2"
19     result = False("a")(null); // result = null
20     @object = new object();
21     result = False(@object)(null); // result = null
22 }
```

Listing 4: Church boolean encoding
The effect that the evaluation of the these combinators have is that the True combinator will return the first argument and the False combinator the second.

5.2.2 Church Boolean arithmetic

The last example of the untyped lambda-calculus in C# shows how the Church booleans can be used to perform boolean logic. In listing 5 the same True and False combinators (simplified with a function type alias) as in Listing 4 are used to define lambda terms that can be used to calculate boolean logic. As an example, the And and Or combinator are shown.

```csharp
// Alias for Func<object, Func<object, object>>
public delegate Func<object, object> Boolean(object @true);

public static class ChurchBoolean
{
    // True := \t.\f.t
    public static Boolean True = @true => @false => @true;

    // False := \t.\f.f
    public static Boolean False = @true => @false => @false;

    // And = a => b => a(b)(False)
    public static Boolean And
        (this Boolean a, Boolean b) =>
        (Boolean)a(b)(new Boolean(False));

    // Or = a => b => a(True)(b)
    public static Boolean Or
        (this Boolean a, Boolean b) =>
        (Boolean)a(new Boolean(True))(b);

    // ...
}
```

Listing 5: Church boolean encoding

Listing 6 shows how to use the combinators to perform boolean logic operations. However, since such an operation yields a function as a result (how it should be in the lambda-calculus), the effect in C# is that there is no way to display this result function. The correct way to deal with this would be to implement a translation between Church Booleans and System.Boolean. However this is out of scope of this document.

```csharp
private static void BooleanLogic()
{
    Boolean True = ChurchBoolean.True;
    Boolean False = ChurchBoolean.False;

    True.And(True);  // true && true
    True.And(False); // true && false
    False.And(True); // false && true
    False.And(False); // false && false
```

20
The untyped lambda-calculus is a very interesting construct. At first when I started researching it, it appeared very strange and unfamiliar. This was mainly because the nature of the lambda calculus differs from everything that I have learned and used so far. It took me quite a while to understand the intention of this concept and how it can be used.

Now that I have a basic understanding of the untyped lambda-calculus it gets much easier to think of areas where it can be quite useful. In my opinion, it is useful to keep this concept in mind, when designing domain specific languages. Because of the mathematical feel of the lambda-calculus it is imaginable to formalize the evaluation of terms of a language in the lambda-calculus.

After all for me the lambda-calculus in modern programming languages is a feature that I really like and that changed the way I write my programs. To be able to yield a function as a function result and to use a function as an argument is a feature of programming languages that I use over and over when I am programming.

It was a very interesting time researching the untyped lambda-calculus. To learn the mathematical foundation behind $\lambda$-expressions in programming languages was what I liked best about the seminar.
References


[lcs14] What is the contribution of lambda calculus to the field of theory of computation?, 2014.


The title image is taken from Dixin’s Blog.[Dix15]