# Summary of "Simply Typed/Untyped Lambda Calculus"

CHRISTOPH MENDE, NIKOLAJ-JENS SCHWARTZ

## 1 THE UNTYPED LAMDBA-CALCULUS

[1]

### 1.1 Introduction

A complex programming language can be understood by formulating it as its essential mechanisms with a collection of derived forms by translating them into the core. The Core language is the lambda-calculus where all computation is reduced to its basic operations of function definiton and application. Since then the lambda-calculus has been widely used in the specification of programming language features, language design and the study of type systems. It can describe computations as a simple programming language and at the same time be a mathematical object about which rigorous statements can be proved. lambda-calculus is a core calculi among others, such as the pi-calculus or the object calculus extract the core features of object-oriented languages. lambda-calculus can be enriched with the ability to handle complex features such as mutable reference cells or nonlocal exception handlung, which would be hard in the core language itself.

### 1.2 Basics

This chapter is about how the lambda-calculus embodies function definition and apllication in the purest possible form. Everything is a function, from the arguments accepted by functions to their results. The syntax of the lambda-calculus with just three sorts of terms. Variable $x$. The Abstraction of variable $x$ from a term $t_1$ written $\lambda x.t_1$ and the application of $t_1$ to $t_2$ written $t_1 t_2$.

*1.2.1 Abstract and Concrete Syntax.* There are two levels of structure. Concrete syntax and abstract syntax. Concrete syntax is the representation a programmer actually reads and writes. Abstract syntax is the simpler internal representation of structures labeled as trees. A lexical analyzer and a parser are needed to transform the concrete to the abstract syntax. While the lexical analyzer turns everything that was written by the programmer into tokens - identifiers, keywords, constants etc. and removes white space, comments etc. the parser transforms the tokens into an abstract syntax tree. Two conventions are adopted when writing lambda-terms in linear form. $stu$ stands for the same tree as $(s\ t)\ u$ and $\lambda x.\lambda y.xyx$ stands for the same tree as $\lambda x.(\lambda y.((xy)x))$

*1.2.2 Varibales and Metavariables.* $t\ s\ u$ will still be used as metavaribles for arbitrary terms. $x\ y\ z$ stand for arbitrary variables. When x, y etc. are used as object-language variables, the context makes clear which is which.

*1.2.3 Scope.* The occurrences of $x$ in $x\ y$ is free, while it is bound in $\lambda x.x$. In another example the first appearance is bound, the second free. $(\lambda x.x)\ x$ A term with no free variables is called *closed* or *combinators*, like $(\lambda x.x)$.

## 2 SIMPLY TYPED LAMBDA-CALCULUS

### 2.1 Function Types

The type system of chapter 8 is extended by primitives of pure lambda-calculus. That is, "typing rules for variables, abstractions, and applications" are added. Variables are simple terms that hold a value. Abstractions are functions

that take a parameter and return a value. Applications are the usage of a function, i.e. a function is called with a parameter.

Since the pure lambda-calculus is turing complete, we need to evaluate the program to determine its return value.

To extend our type system with functions we introduce a new type for functions called $\rightarrow$ so that $\lambda x.t :\rightarrow$. With this new type we can build simple terms like $\lambda x.x$ or compound terms like $if\ true\ then\ (\lambda x.true)\ else\ (\lambda x.\lambda y.y)$).

This approach is too conservative though as $\lambda x.true$ and $\lambda x.\lambda y.y$ have the same type. We need to know the return type to know wether the result is a value or a function. To achieve this, we change our type to $T_1 \rightarrow T_2$ to classify functions that expect $T_1$ and return $T_2$.

With our new $\rightarrow$ we can declare types like $Bool \rightarrow Bool$ for functions that map boolean arguments to boolean results or $(Bool \rightarrow Bool) \rightarrow (Bool \rightarrow Bool)$ for functions that map functions like the aforementioned to functions of the same type.

## 2.2 The Typing Relation

To be able to tell the type of an abstraction like $\lambda x.t$, we need to know what happens when we apply it to some parameter. There are two ways to get the result, we can either analyze the abstraction or annotate it with types. We are going with the annotations and use abstractions like $\lambda x : T_2.t2$ for functions that take a parameter of type $T_2$.

To keep track of the type of variables, we store the name of the variable together with its type as a pair in a set called $\Gamma$.

## REFERENCES

[1] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.