# Introduction to Program Synthesis (SS 2025)

## Chapter 1 - Introduction

Dr. rer. nat. Roman Kalkreuth

Chair for AI Methodology (**AIM**), Department of Computer Science,
RWTH Aachen University, Germany

- Calculus (dt. Kalkül) $\rightarrow$ formal system of rules
- Used to derive statements from given statements (axioms)
- A calculus consists of
  - $\rightsquigarrow$ **Building blocks**
    - Alphabet $\rightarrow$ Symbols, Connectives, structuring signs, ...
  - $\rightsquigarrow$ **Formation rules**
    - Define how building blocks form complex objects or well-formed formulas
    - Analogy to natural language $\rightarrow$ "grammar" of the calculus
  - $\rightsquigarrow$ **Transformation rules**
    - Derivation and deduction rules
    - Transformation to create new objects from them
  - $\rightsquigarrow$ **Axioms**
    - Objects or Expressions
    - Formed according to the formation rules of the calculus
    - Obvious principle

**Example (Chess Calculus)**

Chess game with pieces (axioms) and moves (transformation rules).

- **Formal framework** that can be used in mathematics, logic and programming
- Approach to **systematic solving** problems in certain domain
- Design of suitable **logical frameworks** for **programming languages**
- Examples:
  - **Mathematical** $\rightarrow$ arithmetics, O-Calculus, Stochastic calculus
  - **Logic** $\rightarrow$ propositional calculus
  - **Computation** $\rightarrow$ $\lambda$-Calculus, Turing machine, Plankalkül

- $\lambda$-Calculus $\rightarrow$ minimal model of computation
  - Smallest universal programming language
  - Any computable function can be expressed and evaluated
- Developed by Alonzo Church in the 1930's
  - Published in 1941[Chu85]
- Study of functional computing
- Introduction of a functional notation: $\lambda x.y$
  - $\rightsquigarrow$ Contemporary notation analogy: $x \mapsto y$
  - $\rightsquigarrow$ *Formalisation* of mathematical functions
    - $\mathrm{f(x)} = x^2 \quad \rightsquigarrow \quad x \mapsto x^2,\ \lambda x.x^2$
    - $\mathrm{f(x,y)} = x^2 + y^2 \quad \rightsquigarrow \quad (x, y) \mapsto x^2 + y^2,\ \lambda x.\lambda y.x^2 + y^2$

- Functions are considered expressions $E$
  - Parenthesis can be used for clarity $E \Leftrightarrow (E)$
  - Keywords are only $\lambda$ and the dot
- **Function creation:** Function denotation that has a formal argument $x$ and a functional body $E \rightarrow \lambda x.E$
- **Function application:** Denotation of the application of a function $E_1$ to the argument $E_2 \rightarrow E_1.E_2$
- Syntax of Lambda Calculus:

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

- **Abstraction** $\rightarrow$ anonymous functions
- Single transformation rule $\rightarrow$ **variable substitution**
- Single function definition scheme $\rightarrow$ $\lambda x.x$
  - $\leadsto$ $\lambda$ symbol $\rightarrow$ start of a function expression
  - $\leadsto$ Name after $\lambda$ $\rightarrow$ identifier of the function argument
  - $\leadsto$ Expression after the point $\rightarrow$ function body
- Functions can be applied to expressions $\rightarrow$ $(\lambda x.x)y$
  - $\leadsto$ Evaluation $\rightarrow$ substitution of the argument $x$
  - $\leadsto$ $(\lambda x.x)y = [y/x]\, x = y$
  - $\leadsto$ $[y/x]$ $\rightarrow$ Notation to indicate the substitution of x by y

- Variables can be either free or bound like in math
  - $\leadsto$ **Free variable:** Symbol in an expression that can be substituted
  - $\leadsto$ **Bound variable:** Symbol bound to logical quantifiers or variable-binding operators:
    $\sum_{x=0}^{N}$ $\prod_{x=0}^{\infty}$ $\forall x$ $\exists x$
- $(\lambda x.xy) \rightarrow$ variable x is bound and y is free
- $(\lambda x.x)(\lambda y.yx)$
  - $\leadsto$ $x$ in the first expression is bound to the first $\lambda$
  - $\leadsto$ $y$ in the second expression is bound to the second $\lambda$
  - $\leadsto$ $x$ in the second expression is free

- Functions in standard $\lambda$ calculus are anonymous
  - $\leadsto$ However, capital letters are commonly used to simplify the notation
- For instance, the identity function denoted with `I` serves as a synonym for $(\lambda x.x)$
- **Substitution:** Fundamental mechanism in $\lambda$ calculus
- Computational approach to function composition: $g(x) := (u \circ v)(x) = u(v(x))$

**Example (Identity function)**

- We apply the identity function to itself which is an application:
  - $\rightsquigarrow$ $\mathtt{II} \equiv I_1.I_2 \equiv (\lambda x.x)(\lambda x.x)$
- We can rewrite the expression as:
  - $\rightsquigarrow$ $\mathtt{II} \equiv (\lambda x.x)(\lambda z.z)$
- The identity function when applied to itself leads therefore to:
  - $\rightsquigarrow$ $\mathtt{II} \equiv (\lambda x.x)(\lambda z.z) = [\lambda z.z/x]\,x = \lambda z.z \equiv \mathtt{I}$

- Outermost parentheses can be omitted
  - $\rightsquigarrow$ $(\lambda x.x) \equiv \lambda x.x$
- Function application to arguments is generally left associated
  - $\rightsquigarrow$ $\underbrace{(\lambda x.x)}_{\text{Function}} \quad \underbrace{1}_{\text{Argument}} \equiv \lambda x.x \quad 1$
- There are various notation for substitution
  - $\rightsquigarrow$ $[y/x]\,x$
  - $\rightsquigarrow$ $\{x \leftarrow y\}$
  - $\rightsquigarrow$ $[x := y]$
  - $\rightsquigarrow$ $[x \mapsto a]$

- $\lambda$-calculus knows two computation rules:
- $\alpha$-**equivalence** $\rightarrow$ renaming of variables and parameters
  - $\rightsquigarrow$ $\lambda x.(x\,y) \rightarrow_{\alpha} \lambda x.(x\,z)$
  - $\rightsquigarrow$ $\lambda x.x \equiv \lambda y.y \equiv \lambda z.z$
  - $\rightsquigarrow$ $(\lambda x.x + y) \not\equiv (\lambda y.y + y)$
- $\beta$-**reduction** $\rightarrow$ substitution that is performed in the context of application
  - $\rightsquigarrow$ $(\lambda x.t)a \rightarrow_{\beta} t\,[x \mapsto a]$
  - $\rightsquigarrow$ $(\lambda x.x)\,1 \rightarrow (\lambda 1.1) \rightarrow 1$
- Computing $\lambda$ functions $\rightarrow$ iterative $\beta$-reduction until the normal form is reached which is irreducible
  - $\rightsquigarrow$ $M \rightarrow_{\beta} M_1 \rightarrow_{\beta} M_2 \rightarrow_{\beta} \cdots \rightarrow_{\beta} N \nrightarrow \beta$
  - $\rightsquigarrow$ $\beta$ normal form $N$ $\rightarrow$ no longer possible to apply further arguments
  - $\rightsquigarrow$ $\lambda x.x\ \lambda y.y \rightarrow \lambda y.y$

- $\lambda$ functions have no more than one bound variable and are applied to one argument
  - $\leadsto$ The following function takes two arguments: $\lambda x.(\lambda y.xy)$
  - $\leadsto$ The notation can be simplified as: $\lambda xy.xy$
- Arguments are processed from left to right
  - $\leadsto$ The first argument substitutes the outer $\lambda$
  - $\leadsto$ This process is know as **currying** which means breaking down **a function** that takes **multiple arguments** into **a sequence** of **single argument functions**

$$\begin{array}{r} \lambda x.(\lambda y.xy) \quad 12 \\ [x := 1] \\ \lambda 1.(\lambda y.1y) \quad 1 \\ [y := 2] \\ \lambda 1.(\lambda 2.12) \\ 12 \end{array}$$

- Arithmetic calculations $\rightarrow$ Integral part of a programming language
- We can define a successor function $\Gamma$:
  - $\rightsquigarrow$ $\Gamma(0) = 1$
  - $\rightsquigarrow$ $(\Gamma \circ \Gamma) = \Gamma(\Gamma(0)) = 2$
- **Church numerals** $\rightarrow$ Representation of natural numbers
  - $\rightsquigarrow$ Based on $n$-fold composition function $f^{\circ n} = \underbrace{f \circ f \circ \cdots \circ f}_{n \text{ times}}$
- With $\lambda$-calculus we obtain the following numerals:
  - $\rightsquigarrow$ $0 \equiv \lambda s.(\lambda z.z) = \lambda sz.s$
  - $\rightsquigarrow$ $1 \equiv \lambda sz.s(z)$
  - $\rightsquigarrow$ $2 \equiv \lambda sz.s(s(z))$
  - $\rightsquigarrow$ $3 \equiv \lambda sz.s(s(s(z)))$
- Later more in the framework of the exercise

- Building new terms can be done in two ways:
  - $\leadsto$ $\lambda$-terms $\rightarrow$ build from a variable $x$ and a term $M$ $\rightarrow$ $\lambda x.M$
  - $\leadsto$ Applications $\rightarrow$ build from two terms $M$ and $N$ $\rightarrow$ written as $(M\ N)$
- Terms can be considered or represented as trees
  - $\leadsto$ Inner nodes (non-terminals) are $\lambda$ terms: functions or applications
  - $\leadsto$ Outer nodes (terminals) are variables

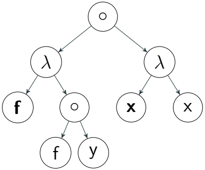


Figure: Tree representation of lambda expression $((\lambda f.(f\ y))\,(\lambda x.x))$

- Pure $\lambda$-calculus is a fundamental ingredient of functional programming languages
  - $+$ reduction strategy
  - $+$ data types
  - $+$ type system
- Building blocks of functional programs
  - $\rightsquigarrow$ Composition of terms

## References I


[Chu85] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton: Princeton University Press, 1985. ISBN: 9781400881932. DOI: doi:10.1515/9781400881932. URL: https://doi.org/10.1515/9781400881932.