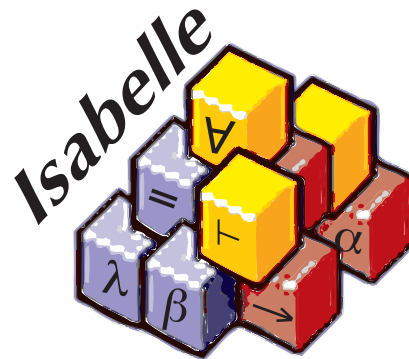


# Introduction to Isabelle

Clemens Ballarin  
Universität Innsbruck



$\lambda$

# Contents

- ▶ Intro & motivation, getting started with Isabelle
- ▶ Foundations & Principles
  - ▶ Lambda Calculus
  - ▶ Types & Classes
  - ▶ Natural Deduction
- ▶ Proof & Specification Techniques
  - ▶ Isar: mathematics style proofs
  - ▶ Inductively defined sets, rule induction
  - ▶ Datatypes, structural induction
  - ▶ Recursive functions & code generation

# $\lambda$ -Calculus

# $\lambda$ -Calculus

## Alonzo Church

- ▶ Lived 1903–1995
- ▶ Supervised people like Alan Turing, Stephen Kleene
- ▶ Famous for Church-Turing thesis,  $\lambda$ -calculus, first undecidability results
- ▶ invented  $\lambda$ -calculus in 1930's



## $\lambda$ -calculus

- ▶ Originally meant as foundation of mathematics
- ▶ Important applications in theoretical computer science
- ▶ Foundation of computability and functional programming

# Untyped $\lambda$ -Calculus

- ▶ Turing-complete model of computation
- ▶ A simple way of writing down functions

Basic intuition:

Instead of  $f(x) = x + 5$   
write  $f = \lambda x. x + 5$

$\lambda x. x + 5$

- ▶ a term
- ▶ a nameless function
- ▶ that adds 5 to its parameter

# Function Application

For applying arguments to functions:

Instead of  $f(x)$   
write  $f\ x$

Example  $(\lambda x. x + 5)\ a$

Evaluating  
(computation) in  $(\lambda x. t)\ a$  replace  $x$  by  $a$  in  $t$

Example  $(\lambda x. x + 5)\ (a + b)$  evaluates to  $(a + b) + 5$

**That's the idea.  
Now formal.**

# Syntax

Terms

$$t ::= v \mid c \mid (t t) \mid (\lambda x. t)$$

$v, x \in V$ ;  $c \in C$ ;  $V, C$  sets of names

- ▶  $v, x$  variables
- ▶  $c$  constants
- ▶  $(t t)$  application
- ▶  $(\lambda x. t)$  abstraction

# Conventions

- ▶ Leave out parentheses where possible
- ▶ List variables instead of multiple  $\lambda$

## Example

Instead of  $(\lambda y. (\lambda x. (x y)))$  write  $\lambda y x. x y$

## Rules

- ▶ List variables:  $\lambda x. (\lambda y. t) = \lambda x y. t$
- ▶ Application binds to the left:  $x y z = (x y) z \neq x (y z)$
- ▶ Abstraction binds to the right:  
 $\lambda x. x y = \lambda x. (x y) \neq (\lambda x. x) y$
- ▶ Leave out outermost parentheses

# Getting Used to the Syntax

## Example

$$\lambda x y z. x z (y z) =$$

$$\lambda x y z. (x z) (y z) =$$

$$\lambda x y z. ((x z) (y z)) =$$

$$\lambda x. \lambda y. \lambda z. ((x z) (y z)) =$$

$$(\lambda x. (\lambda y. (\lambda z. ((x z) (y z))))))$$

# Computation

## Intuition

- ▶ Replace parameter by argument.
- ▶ This is called  $\beta$ -reduction.

## Example

$$\begin{aligned} & (\lambda x y. f (y x)) \ 5 \ (\lambda x. x) \longrightarrow_{\beta} \\ & (\lambda y. f (y \ 5)) \ (\lambda x. x) \longrightarrow_{\beta} \\ & f \ ((\lambda x. x) \ 5) \longrightarrow_{\beta} \\ & f \ 5 \end{aligned}$$

# Defining Computation

## $\beta$ -reduction

$$\begin{array}{l} s \longrightarrow_{\beta} s' \implies (\lambda x. s) t \longrightarrow_{\beta} s[x \leftarrow t] \\ t \longrightarrow_{\beta} t' \implies (s t) \longrightarrow_{\beta} (s' t) \\ s \longrightarrow_{\beta} s' \implies (\lambda x. s) \longrightarrow_{\beta} (\lambda x. s') \end{array}$$

Still to do: define  $s[x \leftarrow t]$

# Defining Substitution

Easy concept. Small problem: variable capture.

**Example**  $(\lambda x. x z)[z \leftarrow x]$

We do **not** want  $(\lambda x. x x)$  as result.

What do we want?

In  $(\lambda y. y z)[z \leftarrow x] = (\lambda y. y x)$  there would be no problem.

So, solution is: rename bound variables.

# Free Variables

## Bound variables

In  $(\lambda x. t)$ ,  $x$  is a **bound** variable.

## Free variables

$FV(t)$  denotes **free** variables of  $t$

$$FV(x) = \{x\} \qquad FV(s\ t) = FV(s) \cup FV(t)$$

$$FV(c) = \{\} \qquad FV(\lambda x. t) = FV(t) \setminus \{x\}$$

## Example

$$FV(\lambda x. (\lambda y. (\lambda x. x) y) y x) = \{y\}$$

Term  $t$  is called **closed** if  $FV(t) = \{\}$

# Substitution

$$\begin{aligned}x [x \leftarrow t] &= t \\y [x \leftarrow t] &= y && \text{if } x \neq y \\c [x \leftarrow t] &= c\end{aligned}$$

$$(s_1 \ s_2) [x \leftarrow t] = (s_1[x \leftarrow t] \ s_2[x \leftarrow t])$$

$$(\lambda x. s) [x \leftarrow t] = (\lambda x. s)$$

$$\begin{aligned}(\lambda y. s) [x \leftarrow t] &= (\lambda y. s[x \leftarrow t]) \\&\text{if } x \neq y \text{ and } y \notin FV(t)\end{aligned}$$

$$\begin{aligned}(\lambda y. s) [x \leftarrow t] &= (\lambda z. s[y \leftarrow z][x \leftarrow t]) \\&\text{if } x \neq y \text{ and } z \notin FV(t) \cup FV(s)\end{aligned}$$

# Substitution Example

$$\begin{aligned} & (x \ (\lambda x. x) \ (\lambda y. z \ x))[x \leftarrow y] \\ = & (x[x \leftarrow y]) \ ((\lambda x. x)[x \leftarrow y]) \ ((\lambda y. z \ x)[x \leftarrow y]) \\ = & y \ (\lambda x. x) \ (\lambda y'. z \ y) \end{aligned}$$

# $\alpha$ -Conversion

Bound names are irrelevant

$\lambda x. x$  and  $\lambda y. y$  denote the same function.

$\alpha$ -conversion

$s =_{\alpha} t$  means  $s = t$  up to renaming of bound variables.

# $\alpha$ -Conversion

**Equality in Isabelle is equality modulo  $\alpha$ -conversion.**

if  $s =_{\alpha} t$  then  $s$  and  $t$  are syntactically equal.

## Examples

$$\begin{aligned} & x (\lambda x y. x y) \\ =_{\alpha} & x (\lambda y x. y x) \\ =_{\alpha} & x (\lambda z y. z y) \\ \neq_{\alpha} & z (\lambda z y. z y) \\ \neq_{\alpha} & x (\lambda x x. x x) \end{aligned}$$

# Back to $\beta$

We have defined  $\beta$ -reduction:  $\longrightarrow_{\beta}$

## Notations and Concepts

- ▶  **$\beta$ -conversion:**  $s =_{\beta} t$  iff  $\exists n. s \longrightarrow_{\beta}^* n \wedge t \longrightarrow_{\beta}^* n$
- ▶  $t$  is **reducible** if there is an  $s$  such that  $t \longrightarrow_{\beta} s$
- ▶  $(\lambda x. s) t$  is called a **redex** (reducible expression)
- ▶  $t$  is reducible iff it contains a redex
- ▶ If it is not reducible,  $t$  is in **normal form**
- ▶  $t$  **has a normal form** if there is an irreducible  $s$  such that  $t \longrightarrow_{\beta}^* s$

# Does Every $\lambda$ -Term Have a Normal Form?

No!

Example

$$\begin{aligned}(\lambda x. x x) (\lambda x. x x) &\longrightarrow_{\beta} \\(\lambda x. x x) (\lambda x. x x) &\longrightarrow_{\beta} \\(\lambda x. x x) (\lambda x. x x) &\longrightarrow_{\beta} \dots\end{aligned}$$

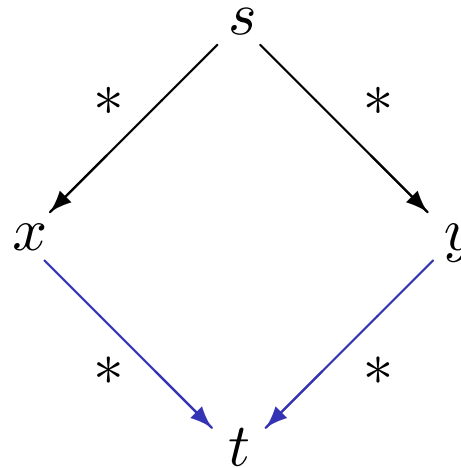
$$\text{(but: } (\lambda x y. y) ((\lambda x. x x) (\lambda x. x x)) \longrightarrow_{\beta} \lambda y. y)$$

$\lambda$ -calculus is **not terminating**.

# $\beta$ -Reduction is Confluent

## Confluence

$$s \longrightarrow_{\beta}^* x \wedge s \longrightarrow_{\beta}^* y \implies \exists t. x \longrightarrow_{\beta}^* t \wedge y \longrightarrow_{\beta}^* t$$



**Order of reduction does not matter for result.**  
Normal forms in  $\lambda$ -calculus are unique.

# $\beta$ -Reduction is Confluent

## Example

$$(\lambda x y. y) ((\lambda x. x x) a) \longrightarrow_{\beta} (\lambda x y. y) (a a) \longrightarrow_{\beta} \lambda y. y$$

$$(\lambda x y. y) ((\lambda x. x x) a) \longrightarrow_{\beta} \lambda y. y$$

# $\eta$ -Reduction

**Another case of trivially equal functions:**  $t = (\lambda x. t x)$

## Example

$$(\lambda x. f x) (\lambda y. g y) \longrightarrow_{\eta} (\lambda x. f x) g \longrightarrow_{\eta} f g$$

- ▶  $\eta$ -reduction is confluent and terminating.
- ▶  $\longrightarrow_{\beta\eta}$  is confluent.  
 $\longrightarrow_{\beta\eta}$  means  $\longrightarrow_{\beta}$  and  $\longrightarrow_{\eta}$  steps are both allowed.
- ▶ Equality in Isabelle is also modulo  $\eta$ -conversion.

## In Fact ...

**Equality in Isabelle is modulo  $\alpha$ -,  $\beta$ -, and  $\eta$ -conversion.**

We will see later why that is possible.

# So, What Can You Do With $\lambda$ -Calculus?

$\lambda$ -calculus is very expressive, you can encode:

- ▶ Logic, set theory
- ▶ Turing machines, functional programs, etc.

## Examples

`true`  $\equiv \lambda x y. x$

`false`  $\equiv \lambda x y. y$

`if`  $\equiv \lambda z x y. z x y$

`if true`  $x y \longrightarrow_{\beta}^* x$

`if false`  $x y \longrightarrow_{\beta}^* y$

Now, not, and, or, etc is easy:

`not`  $\equiv \lambda x. \text{if } x \text{ false true}$

`and`  $\equiv \lambda x y. \text{if } x y \text{ false}$

`or`  $\equiv \lambda x y. \text{if } x \text{ true } y$

# More Examples

## Church Numerals

$$0 \equiv \lambda f x. x$$

$$1 \equiv \lambda f x. f x$$

$$2 \equiv \lambda f x. f (f x)$$

$$3 \equiv \lambda f x. f (f (f x))$$

...

Numeral  $n$  takes arguments  $f$  and  $x$ , applies  $f$   $n$ -times to  $x$ .

$$\text{iszero} \equiv \lambda n. n (\lambda x. \text{false}) \text{true}$$

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

$$\text{add} \equiv \lambda m n. \lambda f x. m f (n f x)$$

# Fixed Points

$$\begin{aligned} & (\lambda x f. f (x x f)) (\lambda x f. f (x x f)) t \longrightarrow_{\beta} \\ & (\lambda f. f ((\lambda x f. f (x x f)) (\lambda x f. f (x x f)) f)) t \longrightarrow_{\beta} \\ & t ((\lambda x f. f (x x f)) (\lambda x f. f (x x f)) t) \end{aligned}$$

$$\begin{aligned} \mu &= (\lambda x f. f (x x f)) (\lambda x f. f (x x f)) \\ \mu t &\longrightarrow_{\beta} t (\mu t) \longrightarrow_{\beta} t (t (\mu t)) \longrightarrow_{\beta} t (t (t (\mu t))) \longrightarrow_{\beta} \dots \end{aligned}$$

$\mu$  is Turing's **fixed point operator**

## Nice, but ...

As a mathematical foundation,  $\lambda$  does not work.

It is inconsistent.

- ▶ **Frege** (Predicate Logic,  $\sim$  1879):  
allows arbitrary quantification over predicates
- ▶ **Russel** (1901): Paradox  $R \equiv \{X \mid X \notin X\}$
- ▶ **Whitehead & Russel** (Principia Mathematica, 1910-1913):  
fix the problem
- ▶ **Church** (1930):  
 $\lambda$ -calculus as logic, true, false,  $\wedge$ , ... as  $\lambda$ -terms

## Problem

with  $\{x \mid P x\} \equiv \lambda x. P x$  and  $x \in M \equiv M x$   
you can write  $R \equiv \lambda x. \text{not } (x x)$   
and get  $(R R) =_{\beta} \text{not } (R R)$

# We Have Learned so Far...

- ▶  $\lambda$ -calculus syntax
- ▶ Free variables, substitution
- ▶  $\beta$ -reduction
- ▶  $\alpha$ - and  $\eta$ -conversion
- ▶  $\beta$ -reduction is confluent
- ▶  $\lambda$ -calculus is very expressive (Turing complete)
- ▶  $\lambda$ -calculus is inconsistent

# Demo

# $\lambda$ -calculus is Inconsistent

We have seen:

Can find term  $R$  such that  $R R =_{\beta} \text{not}(R R)$

There are more terms that do not make sense:

$1\ 2$ , `true false`, etc.

**Solution:** rule out ill-formed terms by using types.  
(Church 1940)

# Introducing Types

Idea: assign a type to each “sensible”  $\lambda$ -term.

## Examples

- ▶ for “term  $t$  has type  $\alpha$ ” write  $t :: \alpha$
- ▶ if  $x$  has type  $\alpha$  then  $\lambda x. x$  is a function from  $\alpha$  to  $\alpha$   
Write:  $(\lambda x. x) :: \alpha \Rightarrow \alpha$
- ▶ for  $s t$  to be sensible:  
 $s$  must be function  
 $t$  must be right type for parameter  
If  $s :: \alpha \Rightarrow \beta$  and  $t :: \alpha$  then  $(s t) :: \beta$

**That's the idea.  
Now formal.**

# Syntax for $\lambda^{\rightarrow}$

**Terms**  $t ::= v \mid c \mid (t t) \mid (\lambda x. t)$   
 $v, x \in V, \quad c \in C, \quad V, C$  sets of names

**Types**  $\tau ::= \mathbf{b} \mid \nu \mid \tau \Rightarrow \tau$   
 $\mathbf{b} \in \{\mathbf{bool}, \mathbf{int}, \dots\}$  base types  
 $\nu \in \{\alpha, \beta, \dots\}$  type variables

$$\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$$

## Contexts

$\Gamma$ : function from variable and constant names to types.

Term  $t$  has type  $\tau$  in context  $\Gamma$ :  $\Gamma \vdash t :: \tau$

# Examples

$$\Gamma \vdash (\lambda x. x) :: \alpha \Rightarrow \alpha$$
$$[y \mapsto \text{int}] \vdash y :: \text{int}$$
$$[z \mapsto \text{bool}] \vdash (\lambda y. y) z :: \text{bool}$$
$$[] \vdash \lambda f x. f x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$$

A term  $t$  is **well-typed** or **type-correct** if there are  $\Gamma$  and  $\tau$  such that  $\Gamma \vdash t :: \tau$

# Type Checking Rules

Variables:  $\Gamma \vdash x :: \Gamma(x)$

Application: 
$$\frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau_1 \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1 t_2) :: \tau_1}$$

Abstraction: 
$$\frac{\Gamma[x \mapsto \tau_1] \vdash t :: \tau_2}{\Gamma \vdash (\lambda x. t) :: \tau_1 \Rightarrow \tau_2}$$

# Example Type Derivation

$$\frac{\frac{[x \mapsto \alpha, y \mapsto \beta] \vdash x :: \alpha}{[x \mapsto \alpha] \vdash \lambda y. x :: \beta \Rightarrow \alpha}}{[] \vdash \lambda x y. x :: \alpha \Rightarrow \beta \Rightarrow \alpha}$$

# More Complex Example

$$\frac{\frac{\frac{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta) \quad \Gamma \vdash x :: \alpha}{\Gamma \vdash f x :: \alpha \Rightarrow \beta}}{\Gamma \vdash f x x :: \beta}}{[f \mapsto \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. f x x :: \alpha \Rightarrow \beta}}{\boxed{\vdash} \vdash \lambda f x. f x x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}$$

$$\Gamma = [f \mapsto \alpha \Rightarrow \alpha \Rightarrow \beta, x \mapsto \alpha]$$

# More General Types

A term can have more than one type.

## Example

$$[] \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool}$$
$$[] \vdash \lambda x. x :: \alpha \Rightarrow \alpha$$

Some types are more general than others:

$$\tau \lesssim \sigma \quad \text{if there is a substitution } S \text{ such that } \tau = S(\sigma)$$

## Examples

$$\text{int} \Rightarrow \text{bool} \lesssim \alpha \Rightarrow \beta \lesssim \beta \Rightarrow \alpha \not\lesssim \alpha \Rightarrow \alpha$$

# Most General Types

**Fact:** each type-correct term has a most general type.

**Formally:**

$$\Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \wedge (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \lesssim \sigma)$$

It can be found by executing the typing rules backwards.

- ▶ **type checking:** checking if  $\Gamma \vdash t :: \tau$  for given  $\Gamma$  and  $\tau$
- ▶ **type inference:** computing  $\Gamma$  and  $\tau$  such that  $\Gamma \vdash t :: \tau$

**Type checking and type inference on  $\lambda^{\rightarrow}$  are decidable.**

# What about $\beta$ -Reduction?

Definition of  $\beta$  reduction stays the same.

**Fact:** Well-typed terms stay well typed-during  $\beta$ -reduction

**Formally:**  $\Gamma \vdash s :: \tau \wedge s \longrightarrow_{\beta} t \implies \Gamma \vdash t :: \tau$

This property is called **subject reduction**.

# What about Termination?

$\beta$ -reduction in  $\lambda \rightarrow$  always terminates.



(Alan Turing, 1942)

- ▶  $=_{\beta}$  **is decidable**  
To decide if  $s =_{\beta} t$ , reduce  $s$  and  $t$  to normal form (always exists, because  $\rightarrow_{\beta}$  terminates), and compare result.
- ▶  $=_{\alpha\beta\eta}$  **is decidable**  
This is why Isabelle can automatically reduce each term to  $\beta\eta$ -normal form.

# What Does This Mean for Expressiveness?

Not all computable functions can be expressed in  $\lambda^{\rightarrow}$ !

How can typed functional languages then be Turing complete?

Fact:

Each computable function can be encoded as closed, type correct  $\lambda^{\rightarrow}$  term using  $Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$  as only constant.

- ▶  $Y$  is Church's fixed point operator
- ▶  $Y\ t =_{\beta} t\ (Y\ t)$
- ▶ used for recursion

# Summary

- ▶ Simply typed lambda calculus  $\lambda^{\rightarrow}$  is consistent.
- ▶  $\beta$ -reduction in  $\lambda^{\rightarrow}$  satisfies subject reduction.
- ▶  $\beta$ -reduction in  $\lambda^{\rightarrow}$  always terminates.
- ▶  $\alpha$ -equivalent terms are equal in Isabelle.
- ▶ Isabelle automatically reduces to  $\beta\eta$ -normal form.