

# introduction & lambda calculus

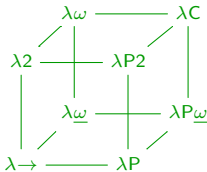
Freek Wiedijk

Type Theory & Coq

2024–2025

Radboud University Nijmegen

September 6, 2024



## organization

### coordinates

---

<https://www.cs.ru.nl/~freek/courses/tt-2024/>

+

Brightspace

### teachers:

- ▶ Freek Wiedijk  
[freek@cs.ru.nl](mailto:freek@cs.ru.nl)
- ▶ Herman Geuvers  
[herman@cs.ru.nl](mailto:herman@cs.ru.nl)
- ▶ Niels van der Weide  
[n.vanderweide@cs.ru.nl](mailto:n.vanderweide@cs.ru.nl)
- ▶ Robbert Krebbers  
[robbert@cs.ru.nl](mailto:robbert@cs.ru.nl)

### teaching assistant:

- ▶ Luko van der Maas  
[luko.vandermaas2@ru.nl](mailto:luko.vandermaas2@ru.nl)

## structure of the course

---

### *first half:*

- ▶ five lectures on the type theory of Coq, by Freek (Fridays)
- ▶ three lectures on metatheory, by Herman (Fridays)
- ▶ Coq practicum (Thursdays)  
→ required, not graded
- ▶ two hour written exam  
→ one third of the final grade

### *second half:*

- ▶ student presentations (Mondays & Fridays)  
45 minutes, in pairs  
→ one third of the final grade
- ▶ Coq project  
→ one third of the final grade

## materials

---

- ▶ Femke van Raamsdonk, VU Amsterdam  
**Logical Verification Course Notes**, 2008
  - ▶ course notes
  - ▶ slides
  - ▶ Coq practicum files
- ▶ Herman Geuvers  
**Introduction to Type Theory**, 2008
  - ▶ summer school lecture notes
  - ▶ slides
  - ▶ some exercises
- ▶ **reading list papers**
- ▶ some supporting documents
  - ▶ Jules Jacobs: **Coq cheat sheet**
  - ▶ examples of **induction/recursion principles**
- ▶ many **old exams**, all with answers

## prerequisites

---

course is self-contained, but...

we will presuppose some basic familiarity with:

- ▶ context-free grammars  
NWI-IPC002 Languages and Automata
- ▶ mathematical logic: natural deduction  
NWI-IPI004 Logic and Applications
- ▶ functional programming  
NWI-IBC040 Functional Programming
- ▶ lambda calculus  
NWI-IBC025 Semantics and Rewriting

as well as some mathematical maturity

## introduction

what is a type?

---

- ▶ an attribute of expressions in a language

← this!

```
int i;  
float pi = 3.14;  
i = 2 * pi;
```

- ▶ something like a set

```
int =  $\{-2^{31}, -2^{31} + 1, \dots, -1, 0, 1, \dots, 2^{31} - 1\}$   
nat =  $\{0, 1, 2, 3, \dots\}$ 
```

## introduction

what is a type?

---

- ▶ an attribute of expressions in a language

← this!

```
int i;  
float pi = 3.14;  
i = 2 * pi;
```

- ▶ something like a set

```
int =  $\{-2^{31}, -2^{31} + 1, \dots, -1, 0, 1, \dots, 2^{31} - 1\}$   
nat =  $\{0, 1, 2, 3, \dots\}$ 
```

but: types do not overlap

the 0 of nat is different from the 0 of int

also: an object has a type

    a type has a kind

        ... but there it stops

## introduction

### what is a type?

---

- ▶ an attribute of expressions in a language


```
int i;  
float pi = 3.14;  
i = 2 * pi;
```

- ▶ something like a set

$$\text{int} = \{-2^{31}, -2^{31} + 1, \dots, -1, 0, 1, \dots, 2^{31} - 1\}$$
$$\text{nat} = \{0, 1, 2, 3, \dots\}$$

but: types do not overlap  
the 0 of nat is different from the 0 of int

also: an object has a type  
a type has a kind  
... but there it stops

 Jules Jacobs @JulesJa... · 30/12/20...  
Replying to @JulesJacobs5 and @IanRay...  
"Types are the things that satisfy the rules  
of type theory." is true, but doesn't help me  
get past syntactic thinking.



## what is type theory?

---

- ▶ **typed lambda calculus**  
≠ untyped lambda calculus (today: recap)
- ▶ a **formal system of datatypes** encoding logic

### Curry-Howard correspondence

pairs in  $A \times B$  correspond to proofs of  $A \wedge B$   
functions in  $A \rightarrow B$  correspond to proofs of  $A \rightarrow B$

- ▶ one of the **logical foundations** for mathematics
  - ▶ set theory
    - ▶ HOL = Higher Order Logic = simple type theory
    - ▶ ZFC = Zermelo-Fraenkel set theory + AC (Axiom of Choice)
  - ▶ type theory
    - ▶ Martin-Löf type theory
    - ▶ CIC = Calculus of Inductive Constructions
  - ▶ category theory
    - ▶ topoi  $\longrightarrow$   $\infty$ -topoi

## the five type theories in this course

---

$\lambda \rightarrow$  = STT

= simple type theory

= simply typed lambda calculus

$\lambda P$  = dependent type theory

$\lambda 2$  = system F

= polymorphic type theory

$\lambda C$  = CC

= Calculus of Constructions

CIC

= Calculus of Inductive Constructions


= the type theory of Coq

## implementations of dependent type theory

---




Coq

INRIA, 1989 

Thierry Coquand, Gérard Huet, Christine Paulin-Mohring, Hugo Herbelin, Matthieu Sozeau



Agda

Chalmers, 1999 

Catarina Coquand, Ulf Norell

→ Cubical Agda



Lean 4

Microsoft Research, 2013 

Leonardo de Moura, Sebastian Ullrich

▶ *other implementations*

Automath, Cubical, Dedukti, Epigram, Idris, Lego, Matita, Nuprl, Plastic, Twelf, ...

## applications of type theory

---

- ▶ advanced functional **programming**

Lisp  $\longrightarrow$  ML  $\longrightarrow$  Haskell  $\longrightarrow$  Agda, Coq

types are **dependent**: carry more information  
'correct by construction'

- ▶ proof **formalization**
  - ▶ verification of programs and other systems
  - ▶ verification of theoretical computer science
  - ▶ verification of mathematics

## applications of type theory

---

- ▶ advanced functional **programming**


Lisp  $\longrightarrow$  ML  $\longrightarrow$  Haskell  $\longrightarrow$  Agda, Coq

types are **dependent**: carry more information  
'correct by construction'

- ▶ proof **formalization**
  - ▶ verification of programs and other systems
  - ▶ verification of theoretical computer science
  - ▶ verification of mathematics
  
- ▶ understanding categorical **foundations** better

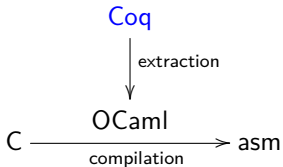
## CompCert

---

- ▶ **CompCert** = verified C compiler  
Xavier Leroy, INRIA 


compiles C to assembly, implemented in Coq  
similar optimization as `gcc -O1`

formal semantics for C and assembly + correctness proof



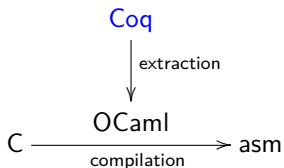
## CompCert



---

- ▶ **CompCert** = verified C compiler  
Xavier Leroy, INRIA 

compiles C to assembly, implemented in Coq  
similar optimization as `gcc -O1`



formal semantics for C and assembly + correctness proof



- ▶  **VST = Verified Software Toolchain**  
Andrew Appel, Princeton   
separation logic, based on CompCert


## Iris

---

Ralf Jung, Zürich , Robbert Krebbers, Nijmegen 

Jacques-Henri Jourdan, Paris , Derek Dreyer, Saarbrücken 

Lars Birkedal, Aarhus 

- ▶ **Ir/s\*** **separation logic** in Coq  
extension of Hoare logic  
pointers in a heap, ownership, concurrency
  - $l \mapsto v$  memory at location  $l$  has value  $v$
  - $P * Q$   $P$  and  $Q$  hold for separate parts of heapprogramming language independent
- ▶  **RustBelt**  
proof (using Iris) of safety and data race freedom of  
**Rust** + some unsafe Rust libraries

→ Robbert Krebbers



## mathematical components

---

Georges Gonthier, Microsoft  → INRIA 

**Ssreflect** proof language for Coq  
**math-comp** mathematical library


- ▶ **four color theorem** (2005)  
every planar graph is four colorable  
proof contains a *huge* computer check
  
- ▶ **Feit-Thompson theorem** = odd order theorem (2012)  
every simple group of odd order is cyclic  
original proof was 255 pages  
→ two full books formalized

## Lean

---

Leonardo de Moura, Microsoft Research → Amazon 

Jeremy Avigad, CMU 

Kevin Buzzard, Imperial College 

Lean = 'Coq#' = Microsoft's Coq clone  
= Coq + Isabelle

- ▶ simpler and *slightly* different type theory  
**extra conveniences**: proof irrelevance, quotient types  
convertibility not transitive, no Subject Reduction
- ▶ implemented in Lean itself (+ small core in C++)  
serious compiler
- ▶ very nice interface based on VS Code
- ▶ **very different user community**: mathematicians!

## mathlib

---

Lean mathematical library  
over a million lines of code

well organized, constantly refactored  
aims to include all undergraduate mathematics (Imperial College)

### large projects:

- ▶ formal definition of **perfectoid spaces**
- ▶ **liquid tensor experiment** (2020–2022)  
challenge by Peter Scholze (Fields medal 2018)
- ▶ polynomial Freiman-Ruzsa (PFR) conjecture over  $\mathbb{F}_2$  (2023)  
**formalization led by Terence Tao** (Fields Medal 2006)  
recent 28 page paper  
formalization took only a few weeks
- ▶ working towards a proof of Fermat's Last Theorem

→ Michail Karatarakis, Freek Wiedijk

# HoTT

---

## Homotopy Type Theory

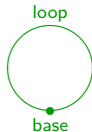
Vladimir Voevodsky (Fields medal 2002), 2006, †2017  

type	~	topological space
function	~	continuous function
equality between points	~	path between points
equality between types	~	equivalence of spaces
$A = B$		$A \simeq B$

- ▶ UA = Univalence Axiom

$$(A = B) \simeq (A \simeq B)$$

- ▶ HITs = Higher Inductive Types  
= types with constructors for equalities



→ Niels van der Weide, Herman Geuvers

## untyped lambda calculus

### lambda abstraction and function application

---

lambda abstraction defines an unnamed function:

$$\lambda x. x^2$$

## untyped lambda calculus

### lambda abstraction and function application

---

lambda abstraction defines an unnamed function:

$\lambda x. x^2$       input:  $x$   
output:  $x^2$

## untyped lambda calculus

### lambda abstraction and function application

---

lambda abstraction defines an unnamed function:

sqr :=  $\lambda x. x^2$       input:  $x$   
output:  $x^2$

## untyped lambda calculus

### lambda abstraction and function application

---

lambda abstraction defines an unnamed function:

$\text{sqr} := \lambda x. x^2$       input:  $x$   
output:  $x^2$

$$\text{sqr}(3) = 9$$

$$\text{sqr } 3 = 9$$



## untyped lambda calculus

### lambda abstraction and function application

---

lambda abstraction defines an unnamed function:

$\text{sqr} := \lambda x. x^2$       input:  $x$   
output:  $x^2$

$$\text{sqr}(3) = 9$$

$$\text{sqr } 3 = 9$$

$$(\lambda x. x^2) 3 = 9$$

## untyped lambda calculus

### lambda abstraction and function application

---

lambda abstraction defines an unnamed function:

$\text{sqr} := \lambda x. x^2$       input:  $x$   
output:  $x^2$

$$\text{sqr}(3) = 9$$

$$\text{sqr } 3 = 9$$

lambda abstraction

$$\overbrace{(\lambda x. x^2)} \text{ } 3 = 9$$

## untyped lambda calculus

### lambda abstraction and function application

---

lambda abstraction defines an unnamed function:

$$\text{sqr} := \lambda x. x^2 \quad \begin{array}{l} \text{input: } x \\ \text{output: } x^2 \end{array}$$

$$\text{sqr}(3) = 9$$

$$\text{sqr } 3 = 9$$

lambda abstraction

$$\underbrace{(\lambda x. x^2)}_{\text{function application}} 3 = 9$$

function application

## syntax versus semantics

---

$\lambda x. x$  a string of six symbols

## syntax versus semantics

---

$\lambda x. x$       a string of six symbols  
( $\lambda x . x$ )

## syntax versus semantics

---

$\lambda x. x$       a string of six symbols  
( $\lambda x . x$ )

$\llbracket \lambda x. x \rrbracket$       a function (the identity function)

## syntax versus semantics

---

$\lambda x. x$       a string of six symbols  
( $\lambda x. x$ )

$\llbracket \lambda x. x \rrbracket$       a function (the identity function)

no semantics of *untyped* lambda calculus in this course  
not trivial!

→ [NWI-IMC011 Semantics and Domain Theory](#)

## examples of untyped lambda terms

---

$x$	$\lambda n f x. f(n f x)$
$xx$	$\lambda m n f x. m f(n f x)$
$xy$	$\lambda m n f x. m(n f)x$
$\lambda x. x$	$\lambda m n f x. n m f x$
$\lambda x. y$	$\lambda x. xx$
$\lambda xy. x$	$(\lambda x. xx)(\lambda x. xx)$
$\lambda xy. y$	$(\lambda x. f(xx))(\lambda x. f(xx))$
$\lambda xyz. xz(yz)$	$\lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$
$\lambda fxy. fyx$	$\lambda x f. f(xxf)$
$\lambda fx. fxx$	$(\lambda x f. f(xxf))(\lambda x f. f(xxf))$
$\lambda fgx. f(gx)$	$\lambda x. x(\lambda xyz. xz(yz))(\lambda xy. x)$
$\lambda fx. x$	
$\lambda fx. fx$	
$\lambda fx. f(fx)$	
$\lambda fx. f(f(fx))$	
$\vdots$	



## variables

---

the set of variables is called **Var**

it does not matter what this set is,  
as long as it is countably infinite

for the formal definition of untyped lambda terms we will take

$$\text{Var} = \{x, x', x'', x''', \dots\}$$

but we will write these as

$x,$   
 $x', x'', x''', \dots$   
 $x_0, x_1, x_2, x_3, \dots$   
 $y, z, u, v, w, n, m, f, g, h, \dots$   
 $y', y'', y''', \dots$   
 $y_0, y_1, y_2, y_3, \dots$   
 $\dots$

## alpha equivalence

---

$$\lambda x. x^2 \not\equiv \lambda y. y^2$$

$$\lambda x. x^2 =_{\alpha} \lambda y. y^2$$

$$M \equiv N$$

$M$  and  $N$  are equal as strings

$$M =_{\alpha} N$$

‘names of variables bound by lambdas do not matter’

in practice we only consider lambda terms **modulo**  $=_{\alpha}$

## alpha equivalence

---

$$\lambda x. x^2 \not\equiv \lambda y. y^2$$

$$\lambda x. x^2 =_{\alpha} \lambda y. y^2$$

$$x^2 \not\equiv y^2$$

$$x^2 \not\equiv_{\alpha} y^2$$

in the first case the variables  $x$  and  $y$  are **bound**

in the second case the variables  $x$  and  $y$  are **free**

$FV(M)$  is the set of free variables in the term  $M$

$$M \equiv N$$

$M$  and  $N$  are equal as strings

$$M =_{\alpha} N$$

‘names of variables bound by lambdas do not matter’

in practice we only consider lambda terms **modulo**  $=_{\alpha}$

## formal definition of untyped lambda terms

---

the set of untyped lambda terms  $\Lambda$  is the smallest set which

- ▶ contains all **variables**

if  $x \in \text{Var}$  then  $x \in \Lambda$

- ▶ is closed under **function application**

if  $F, M \in \Lambda$  then also  $(FM) \in \Lambda$

- ▶ is closed under **lambda abstraction**

if  $x \in \text{Var}$  and  $M \in \Lambda$  then  $(\lambda x. M) \in \Lambda$

## context-free grammar of untyped lambda terms

---

the set of variables  $\text{Var}$   
and the set of untyped lambda terms  $\Lambda$   
are sets of strings over the alphabet

$$\{\lambda, ., (, ), x, '\}$$

$$\begin{array}{ll} x ::= x \mid x' & \text{Var} \\ M ::= x \mid (MM) \mid (\lambda x. M) & \Lambda \end{array}$$

$$\lambda fxy. fyx$$

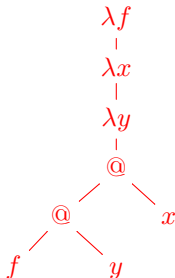
is the  $=_\alpha$ -equivalence class of the 28-symbol string

$$(\lambda x. (\lambda x'. (\lambda x''. ((xx'') x')))) \in \Lambda$$

## abstract syntax trees

---

the parentheses in the grammar are for non-ambiguity

$$\lambda fxy. fyx$$
$$(\lambda f.(\lambda x.(\lambda y.((fy)x))))$$
$$(\lambda x''.(\lambda x.(\lambda x'.((x''x')x))))$$


## notation

---

- ▶ parentheses may be omitted or added
- ▶ lambda abstraction binds more weakly than application:

$$\lambda x. yz \equiv ((\lambda x. y)z) \text{ or } (\lambda x. (yz))$$

- ▶ application associates to the left:

$$xyz \equiv ((xy)z) \text{ or } (x(yz))$$

## notation

---

- ▶ parentheses may be omitted or added
- ▶ lambda abstraction binds more weakly than application:

$$\lambda x. yz \equiv ((\lambda x. y)z) \text{ or } (\lambda x. (yz))$$

- ▶ application associates to the left:

$$xyz \equiv ((xy)z) \text{ or } (x(yz))$$



## notation

---

- ▶ parentheses **may be omitted or added**
- ▶ lambda abstraction **binds more weakly** than application:

$$\lambda x. yz \equiv ((\lambda x. y)z) \text{ or } (\lambda x. (yz))$$

- ▶ application **associates to the left**:

$$xyz \equiv ((xy)z) \text{ or } (x(yz))$$

## notation

---

- ▶ parentheses may be omitted or added
- ▶ lambda abstraction binds more weakly than application:

$$\lambda x. yz \equiv ((\lambda x. y)z) \text{ or } (\lambda x. (yz))$$

- ▶ application associates to the left:

$$xyz \equiv ((xy)z) \text{ or } (x(yz))$$

Curried function with three arguments applied to three values:

$$\begin{aligned} & (\lambda xyz. M)abc \\ & \quad \quad \quad \equiv \\ & (((\lambda x. (\lambda y. (\lambda z. M))) a) b) c \end{aligned}$$

what is this  $x^2$  anyway?

---

in untyped lambda calculus **everything** is a function  
there is **only** lambda abstraction and function application

## what is this $x^2$ anyway?

---

in untyped lambda calculus **everything** is a function  
there is **only** lambda abstraction and function application

- ▶ **numbers** are functions

$$0 = \lambda f x. x$$

$$7 = \lambda f x. f(f(f(f(f(f(fx))))))$$

$$x^2 = \lambda y z. x(xy)z$$

- ▶ **Booleans** are functions

$$\text{false} = \lambda xy. y$$

$$\text{true} = \lambda xy. x$$

- ▶ in untyped lambda calculus the elements of **all datatypes** are coded as functions

## computation

### beta reduction

---

'compute' the value of

$$(\lambda x. x^2) (y + 1)$$

substitute  $(y + 1)$  for the  $x$  under the lambda:

$$(\lambda x. x^2) (y + 1) \rightarrow_{\beta} (y + 1)^2$$

## computation

### beta reduction

---

'compute' the value of

$$(\lambda x. x^2) (y + 1)$$

substitute  $(y + 1)$  for the  $x$  under the lambda:

$$(\lambda x. x^2) (y + 1) \rightarrow_{\beta} (y + 1)^2$$

general form of the beta rule:

$$\underbrace{(\lambda x. M) N}_{\text{redex}} \rightarrow_{\beta} M[x := N]$$

substitution operation on terms comes later:

$$M[x := N]$$

## three relations between terms

---



$$M \rightarrow_{\beta} N$$

one-step reduction  
subterms also can be redexes



$$M \twoheadrightarrow_{\beta} N$$

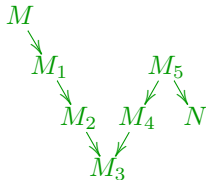
$$M \rightarrow_{\beta} M_1 \rightarrow_{\beta} M_2 \rightarrow_{\beta} \cdots \rightarrow_{\beta} N$$

many-step reduction  
zero, one or more steps



$$M =_{\beta} N$$

**convertible** = computationally equal  
zero, one or more steps in both directions  
smallest equivalence relation containing  $\rightarrow_{\beta}$



## example reduction

---

$$I = \lambda x. x$$

$$K = \lambda xy. x$$

$$\omega = \lambda x. xx$$

$$\Omega = \omega\omega$$

$$K I \Omega \rightarrow_{\beta} I$$

$$\begin{array}{c} K I \Omega \\ \equiv \\ (\lambda xy. x)(\lambda z. z) \Omega \\ \downarrow_{\beta} \\ (\lambda y. (\lambda z. z)) \Omega \\ \equiv \\ (\lambda yz. z) \Omega \\ \downarrow_{\beta} \\ \lambda z. z \\ \equiv \\ I \end{array}$$



## example reduction

---

$$I = \lambda x. x$$

$$K = \lambda xy. x$$

$$\omega = \lambda x. xx$$

$$\Omega = \omega\omega$$

$$KI\Omega \rightarrow_{\beta} I$$

$$\begin{array}{c} KI\Omega \\ \equiv \\ (\lambda xy. x)(\lambda z. z)\Omega \\ \downarrow \beta \\ (\lambda y. (\lambda z. z))\Omega \\ \equiv \\ (\lambda yz. z)\Omega \\ \downarrow \beta \\ \lambda z. z \\ \equiv \\ I \end{array}$$

## example reduction

---

$$I = \lambda x. x$$

$$K = \lambda xy. x$$

$$\omega = \lambda x. xx$$

$$\Omega = \omega\omega$$

$$KI\Omega \rightarrow_{\beta} I$$

$$\begin{array}{c} KI\Omega \\ \equiv \\ (\lambda xy. x)(\lambda z. z) \Omega \\ \downarrow_{\beta} \\ (\lambda y. (\lambda z. z)) \Omega \\ \equiv \\ (\lambda yz. z) \Omega \\ \downarrow_{\beta} \\ \lambda z. z \\ \equiv \\ I \end{array}$$

beware of the brackets!

---

$$(\lambda xy. x)(\lambda z. z) \Omega$$

beware of the brackets!

---

$$(\lambda xy. x)(\lambda z. z) \Omega$$

beta redex?

## beware of the brackets!

---

$$(\lambda xy. x) \underbrace{(\lambda z. z)}_{\text{beta redex?}} \Omega$$

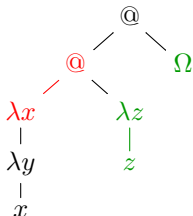
$$\underbrace{((\lambda xy. x) (\lambda z. z))}_{\text{not a beta redex!}} \Omega$$

## beware of the brackets!

---

$$(\lambda xy. x) \underbrace{(\lambda z. z) \Omega}_{\text{beta redex?}}$$

$$\underbrace{((\lambda xy. x) (\lambda z. z)) \Omega}_{\text{not a beta redex!}}$$



## avoiding variable capture by renaming

---

$$\begin{aligned}\omega &= \lambda x. xx \\ 1 &= \lambda fx. fx\end{aligned}$$

$$\omega 1 \twoheadrightarrow_{\beta} 1$$

$$\begin{aligned}& \omega 1 \\ & \equiv \\ & (\lambda z. zz)(\lambda fx. fx) \\ & \downarrow_{\beta} \\ & (\lambda fx. fx)(\lambda fx. fx) \\ & \downarrow_{\beta} \\ & \lambda x. (\lambda fx. fx) x \not\rightarrow_{\beta} \lambda x. (\lambda x. xx) \\ & \equiv \\ & \lambda x. (\lambda fx'. fx') x \\ & \downarrow_{\beta} \\ & \lambda x. \lambda x'. xx' \\ & \equiv \\ & \lambda xx'. xx' \\ & \equiv \\ & 1\end{aligned}$$

## avoiding variable capture by renaming

---

$$\omega = \lambda x. xx$$

$$1 = \lambda fx. fx$$

$$\omega 1 \twoheadrightarrow_{\beta} 1$$

$$\begin{aligned} & \omega 1 \\ & \equiv \\ & (\lambda z. zz)(\lambda fx. fx) \\ & \downarrow_{\beta} \\ & (\lambda fx. fx)(\lambda fx. fx) \\ & \downarrow_{\beta} \\ & \lambda x. (\lambda fx. fx) x \not\rightarrow_{\beta} \lambda x. (\lambda x. xx) \\ & \equiv \\ & \lambda x. (\lambda fx'. fx') x \\ & \downarrow_{\beta} \\ & \lambda x. \lambda x'. xx' \\ & \equiv \\ & \lambda xx'. xx' \\ & \equiv \\ & 1 \end{aligned}$$



## avoiding variable capture by renaming

---

$$\begin{aligned}\omega &= \lambda x. xx \\ 1 &= \lambda fx. fx\end{aligned}$$

$$\omega 1 \twoheadrightarrow_{\beta} 1$$

$$\begin{aligned}\omega 1 & \\ \equiv & \\ (\lambda z. zz)(\lambda fx. fx) & \\ \downarrow_{\beta} & \\ (\lambda fx. fx)(\lambda fx. fx) & \\ \downarrow_{\beta} & \\ \lambda x. (\lambda fx. fx) x \not\rightarrow_{\beta} \lambda x. (\lambda x. xx) & \\ \equiv & \\ \lambda x. (\lambda fx'. fx') x & \\ \downarrow_{\beta} & \\ \lambda x. \lambda x'. xx' & \\ \equiv & \\ \lambda xx'. xx' & \\ \equiv & \\ 1 & \end{aligned}$$

## avoiding variable capture by renaming

---

$$\omega = \lambda x. xx$$

$$1 = \lambda fx. fx$$

$$\omega 1 \twoheadrightarrow_{\beta} 1$$

$$\begin{aligned} & \omega 1 \\ & \equiv \\ & (\lambda z. zz)(\lambda fx. fx) \\ & \downarrow_{\beta} \\ & (\lambda fx. fx)(\lambda fx. fx) \\ & \downarrow_{\beta} \\ & \lambda x. (\lambda fx. fx) x \not\rightarrow_{\beta} \lambda x. (\lambda x. xx) \\ & \equiv \\ & \lambda x. (\lambda fx'. fx') x \\ & \downarrow_{\beta} \\ & \lambda x. \lambda x'. xx' \\ & \equiv \\ & \lambda xx'. xx' \\ & \equiv \\ & 1 \end{aligned}$$

## avoiding variable capture by renaming

---

$$\omega = \lambda x. xx$$

$$1 = \lambda fx. fx$$

$$\omega 1 \twoheadrightarrow_{\beta} 1$$

$$\begin{aligned} & \omega 1 \\ & \equiv \\ & (\lambda z. zz)(\lambda fx. fx) \\ & \downarrow_{\beta} \\ & (\lambda fx. fx)(\lambda fx. fx) \\ & \downarrow_{\beta} \\ & \lambda x. (\lambda fx. fx) x \not\rightarrow_{\beta} \lambda x. (\lambda x. xx) \\ & \equiv \\ & \lambda x. (\lambda fx'. fx') x \\ & \downarrow_{\beta} \\ & \lambda x. \lambda x'. xx' \\ & \equiv \\ & \lambda xx'. xx' \\ & \equiv \\ & 1 \end{aligned}$$

## example with more than one reduction path

---

$$I = \lambda x. x$$

$$\underline{IM} \equiv (\lambda x. x)M \rightarrow_{\beta} M$$

the red lines are not part of the syntax  
they just indicate where the redexes are

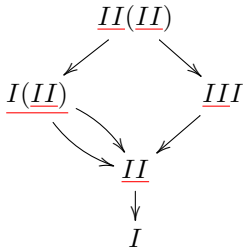
## example with more than one reduction path

---

$$I = \lambda x. x$$

$$\underline{IM} \equiv (\lambda x. x)M \rightarrow_{\beta} M$$

the red lines are not part of the syntax  
they just indicate where the redexes are



$$II(II) \twoheadrightarrow_{\beta} I$$

## wrapping up

### Currying revisited

---

- ▶ traditional mathematics:

$$\begin{aligned} f(x) \\ f(g(x)) \\ h(x, y) \end{aligned}$$

$$h : A \times B \rightarrow C$$

- ▶ lambda calculus and type theory:

$$\begin{aligned} fx \\ f(gx) \\ hxy \quad \equiv \quad (hx)y \end{aligned}$$

$$h : A \rightarrow B \rightarrow C$$

$$hx : B \rightarrow C$$

$$hxy : C$$

## partial function application

---

$$\text{add} = \lambda xy. x + y = \lambda x. (\lambda y. x + y)$$

$$\text{add } 3 = \lambda y. 3 + y$$

$$\text{add } 3 \ 4 = 3 + 4 = 7$$

$$\text{add} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$

$$\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$$

## substitution more formally

---

recursive definition of substitution

the terms are modulo  $=_\alpha$

$$x[x := N] \equiv N$$

$$y[x := N] \equiv y \qquad y \neq x$$

$$(M_1 M_2)[x := N] \equiv (M_1[x := N] M_2[x := N])$$

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

$$(\lambda y'. M)[x := N] \equiv (\lambda y'. M[x := N]) \qquad y' \neq x, y' \notin \text{FV}(N)$$

if you want to be specific, you can let  $y'$  be the first variable from  $\text{Var} \setminus (\{x\} \cup \text{FV}(M) \cup \text{FV}(N))$

in practice, we always work in  $\Lambda / =_\alpha$



## fast-and-loose context-free grammars

---

$$x ::= x \mid x'$$
$$M ::= x \mid (MM) \mid (\lambda x. M)$$

$$M, N ::= x \mid MN \mid \lambda x. M$$

in this course from now on:

- ▶ no parentheses in grammars  
imagine them being there  
or imagine  $\Lambda$  to consist of abstract syntax trees
- ▶ no grammar rules for the variables  
imagine them being there  
or consider sets like  $\text{Var}$  to be a parameter of the definition
- ▶ multiple names for the same non-terminal

## recap

---

- ▶ a set of lambda terms as strings called  $\Lambda$
- ▶ relations  $\equiv, =_{\alpha}, \rightarrow_{\beta}, \twoheadrightarrow_{\beta}, =_{\beta}$
- ▶ Curried functions
- ▶ fast-and-loose context-free grammars

## recap

---

- ▶ a set of lambda terms as strings called  $\Lambda$
- ▶ relations  $\equiv, =_{\alpha}, \rightarrow_{\beta}, \twoheadrightarrow_{\beta}, =_{\beta}$
- ▶ Curried functions
- ▶ fast-and-loose context-free grammars

## homework for next Thursday:

- ▶ install Coq on your computer
- ▶ download the Coq practicum files

## T-shirt

### a lambda calculus evaluator

---

$$U_k^i \equiv \lambda x_1 \dots x_k. x_i$$
$$\langle M_1, \dots, M_k \rangle \equiv \lambda z. z M_1 \dots M_k$$

$$\langle \langle \mathbf{K}, \mathbf{S}, \mathbf{C} \rangle \rangle \ulcorner M \urcorner \twoheadrightarrow M$$

$$\ulcorner x \urcorner \equiv \lambda e. e U_3^1 x e$$
$$\ulcorner PQ \urcorner \equiv \lambda e. e U_3^2 \ulcorner P \urcorner \ulcorner Q \urcorner e$$
$$\ulcorner \lambda x. P \urcorner \equiv \lambda e. e U_3^3 (\lambda x. \ulcorner P \urcorner) e$$

$$\mathbf{K} \equiv \lambda xy. x$$

$$\mathbf{S} \equiv \lambda xyz. xz(yz)$$

$$\mathbf{C} \equiv \lambda xyz. xzy$$

## table of contents

contents

---

organization

introduction

untyped lambda calculus

computation

wrapping up

T-shirt

table of contents