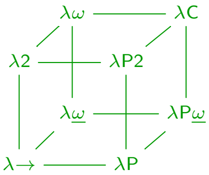
introduction & lambda calculus

Freek Wiedijk

Type Theory & Coq
2023–2024
Radboud University Nijmegen

September 8, 2023



organization

coordinates

https://www.cs.ru.nl/~freek/courses/tt-2023/

$+$

Brightspace

teachers:

- Freek Wiedijk
  freek@cs.ru.nl
- Herman Geuvers
  herman@cs.ru.nl
- Robbert Krebbers
  robbert@cs.ru.nl
- Marc Hermes
  marc.hermes@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 (Mondays)
  $\longrightarrow$ required, not graded
- three hour written exam
  $\longrightarrow$ one third of the final grade

second half:

- student presentations (Mondays & Fridays)
  45 minutes, in pairs
  $\longrightarrow$ one third of the final grade
- Coq project
  $\longrightarrow$ 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...

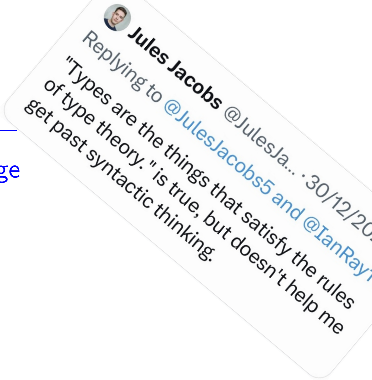
we will presuppose some basic familiarity with:

- 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

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

- something like a set

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

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

what is type theory?

- typed lambda calculus
  $\neq$ untyped lambda calculus (today: recap)

- logic encoded as a formal system of datatypes

  Curry-Howard correspondence

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

- 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 $\longrightarrow$ ∞-topoi

the five type theories in this course

$$\lambda{\to} = \text{STT}$$
$$= \text{simple type theory}$$
$$= \text{simply typed lambda calculus}$$

$$\lambda P = \text{dependent type theory}$$

$$\lambda 2 = \text{system F}$$
$$= \text{polymorphic type theory}$$

$$\lambda C = \text{CC}$$
$$= \text{Calculus of Constructions}$$

CIC

$$= \text{Calculus of Inductive Constructions}$$
$$= \text{the type theory of Coq}$$

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

  $\longrightarrow$ 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

$$\text{Lisp} \longrightarrow \text{ML} \longrightarrow \text{Haskell} \longrightarrow \text{Agda, Coq}$$

  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

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

$$\begin{array}{c} \text{Coq} \\ \Big\downarrow \text{extraction} \\ \text{C} \xrightarrow[\text{compilation}]{\text{OCaml}} \text{asm} \end{array}$$

- VST = Verified Software Toolchain
  Andrew Appel, Princeton

  separation logic, based on CompCert

Ralf Jung, Zürich, Robbert Krebbers, Nijmegen
Jacques-Henri Jourdan, Paris, Derek Dreyer, Saarbrücken
Lars Birkedal, Aarhus

- Iris 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 heap

  programming language independent

- RustBelt

  proof (using Iris) of safety and data race freedom of
  Rust + some unsafe Rust libraries

$\longrightarrow$ Robbert Krebbers

mathematical components

Georges Gonthier, Microsoft $\longrightarrow$ 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

  $\longrightarrow$ two full books formalized

Homotopy Type Theory

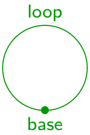
Vladimir Voevodsky (Fields medal 2002), 2006, †2017 

type	$\sim$	topological space
function	$\sim$	continuous function
equality between points	$\sim$	path between points
equality between types	$\sim$	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



$\longrightarrow$ Niels van der Weide, Herman Geuvers

Lean

Leonardo de Moura, Microsoft Research $\longrightarrow$ Amazon
Kevin Buzzard, Imperial College
Jeremy Avigad, CMU

$$\begin{aligned} \text{Lean} &= \text{`Coq\#'} = \text{Microsoft's Coq clone} \\ &= \text{Coq} + \text{Isabelle} \end{aligned}$$

- 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)
- working towards a proof of Fermat's Last Theorem

$\longrightarrow$ Michail Karatarakis, Freek Wiedijk

lambda abstraction and function application

lambda abstraction defines an unnamed function:

$$\text{sqr} := \lambda x.\, x^2 \qquad \begin{array}{rl} \text{input:} & x \\ \text{output:} & x^2 \end{array}$$

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

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

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

$\lambda x.\, x$ $(\, \lambda\, \mathsf{x}\, .\, \mathsf{x}\,)$	a string of six symbols
$[\![\lambda x.\, x]\!]$	a function (the identity function)

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

$\longrightarrow$ NWI-IMC011 Semantics and Domain Theory

examples of untyped lambda terms

$$
\begin{array}{cc}
x & \lambda nfx.\, f(nfx) \\
xx & \lambda mnfx.\, mf(nfx) \\
xy & \lambda mnfx.\, m(nf)x \\
\lambda x.\, x & \lambda mnfx.\, nmfx \\
\lambda x.\, y & \lambda x.\, xx \\
\lambda xy.\, x & (\lambda x.\, xx)(\lambda x.\, xx) \\
\lambda xy.\, y & (\lambda x.\, f(xx))(\lambda x.\, f(xx)) \\
\lambda xyz.\, xz(yz) & \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx)) \\
\lambda fxy.\, fyx & \lambda xf.\, f(xxf) \\
\lambda fx.\, fxx & (\lambda xf.\, f(xxf))(\lambda xf.\, f(xxf)) \\
\lambda fgx.\, f(gx) & \lambda x.\, x(\lambda xyz.\, xz(yz))(\lambda xy.\, x) \\
\lambda fx.\, x & \\
\lambda fx.\, fx & \\
\lambda fx.\, f(fx) & \\
\lambda fx.\, f(f(fx)) & \\
\vdots &
\end{array}
$$

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

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

but we will write these as

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

alpha equivalence

$$\lambda x.\, x^2 \not\equiv \lambda y.\, y^2$$
$$\lambda x.\, x^2 =_\alpha \lambda y.\, y^2$$

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

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 Λ is the smallest set which

- contains all variables

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

- is closed under function application

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

- is closed under lambda abstraction

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

context-free grammar of untyped lambda terms

the set of variables Var
and the set of untyped lambda terms Λ
are sets of strings over the alphabet

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

$$
\begin{array}{rcll}
x & ::= & \mathsf{x} \mid x' & \qquad\qquad \text{Var} \\
M & ::= & x \mid (M\,M) \mid (\lambda x.\,M) & \qquad\qquad \Lambda
\end{array}
$$

$$\lambda fxy.\,fyx$$

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

$$(\lambda\mathsf{x}.(\lambda\mathsf{x}'.(\lambda\mathsf{x}''.((\mathsf{x}\mathsf{x}'')\,\mathsf{x}')))) \in \Lambda$$

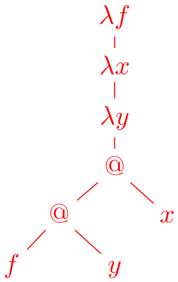
abstract syntax trees

the parentheses in the grammar are for non-ambiguity

$$\lambda fxy.\, fyx$$

$$(\lambda f.(\lambda x.(\lambda y.((fy)x))))$$

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



notation

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

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

- application associates to the left:

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

Curried function with three arguments applied to three values:

$$(\lambda xyz.M)abc$$
$$|||$$
$$((((\lambda x.(\lambda y.(\lambda z.\, M)))\, a)\, b)\, c)$$

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

$$0 = \lambda f x.\, x$$
$$7 = \lambda f x.\, f(f(f(f(f(f(fx))))))$$
$$x^2 = \lambda yz.x(xy)z$$

- 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

beta reduction

'compute' the value of

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

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

$$(\lambda x.\, x^2)\,(y+1) \;\to_\beta\; (y+1)^2$$

general form of the beta rule:

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

substitution operation on terms comes later:

$$M[x := N]$$

three relations between terms

- $$M \to_\beta N$$

  one-step reduction
  subterms also can be redexes

- $$M \twoheadrightarrow_\beta N$$

  $$M \to_\beta M_1 \to_\beta M_2 \to_\beta \cdots \to_\beta N$$

  many-step reduction
  zero, one or more steps

- $$M =_\beta N$$

  $$\begin{tikzcd}[column sep=tiny]
M \arrow[dr] & & & & & & \\
 & M_1 \arrow[dr] & & & & M_5 \arrow[dl] \arrow[dr] & \\
 & & M_2 \arrow[dr] & & M_4 \arrow[dl] & & N \\
 & & & M_3 & & &
\end{tikzcd}$$

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

example reduction

$I = \lambda x.\, x$
$K = \lambda xy.\, x$
$\omega = \lambda x.\, xx$
$\Omega = \omega\omega$

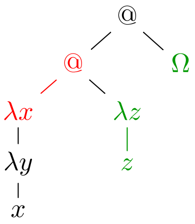
$$
\begin{array}{c}
KI\Omega \\
\equiv \\
(\lambda xy.\, x)(\lambda z.\, z)\,\Omega \\
\downarrow_\beta \\
(\lambda y.\, (\lambda z.\, z))\,\Omega \\
\equiv \\
(\lambda yz.\, z)\,\Omega \\
\downarrow_\beta \\
\lambda z.\, z \\
\equiv \\
I
\end{array}
$$

$$KI\Omega \twoheadrightarrow_\beta I$$

beware of the brackets!

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

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



avoiding variable capture by renaming

$\omega = \lambda x.\, xx$
$1 = \lambda fx.\, fx$

$\omega 1 \twoheadrightarrow_\beta 1$

$$\begin{array}{c}
\omega 1 \\
\equiv \\
(\lambda z.\, zz)(\lambda fx.\, fx) \\
\downarrow_\beta \\
(\lambda fx.\, fx)(\lambda fx.\, fx) \\
\downarrow_\beta \\
\lambda x.\, (\lambda fx.\, fx)\, x \quad \not\to_\beta \quad \lambda x.\, (\lambda x.\, xx) \\
=_\alpha \\
\lambda x.\, (\lambda fx'.\, fx')\, x \\
\downarrow_\beta \\
\lambda x.\, \lambda x'.\, xx' \\
\equiv \\
\lambda xx'.\, xx' \\
=_\alpha \\
1
\end{array}$$

example with more than one reduction path

$I = \lambda x.\, x$

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

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

$$\begin{tikzcd}
& \underline{II}(\underline{II}) \arrow[dl] \arrow[dr] & \\
\underline{I(\underline{II})} \arrow[dr, bend right] \arrow[dr, bend left] & & \underline{II}I \arrow[dl] \\
& \underline{II} \arrow[d] & \\
& I &
\end{tikzcd}$$

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

Currying revisited

- traditional mathematics:

$$f(x)$$
$$f(g(x))$$
$$h(x, y)$$

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

- lambda calculus and type theory:

$$fx$$
$$f(gx)$$
$$hxy \qquad \equiv (hx)y$$

$$\begin{aligned} h &: A \to B \to C \\ hx &: B \to C \\ hxy &: C \end{aligned}$$

partial function application

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

$$\mathsf{add}\ 3 = \lambda y.\, 3 + y$$

$$\mathsf{add}\ 3\,4 = 3 + 4 = 7$$

$$\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

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

recursive definition of substitution
the terms are modulo $=_\alpha$

$$
\begin{aligned}
x[x := N] &\equiv N \\
y[x := N] &\equiv y && y \neq x \\
(M_1 M_2)[x := N] &\equiv (M_1[x := N] M_2[x := N]) \\
(\lambda x.\, M)[x := N] &\equiv (\lambda x.\, M) \\
(\lambda y'.\, M)[x := N] &\equiv (\lambda y'.\, M[x := N]) && y' \neq x,\ y' \notin \mathsf{FV}(N)
\end{aligned}
$$

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

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

fast-and-loose context-free grammars

$$x ::= \mathsf{x} \mid x'$$
$$M ::= x \mid (MM) \mid (\lambda x.\, M)$$

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

in this course from now on:

- no parentheses in grammars

  imagine them being there
  or imagine Λ to consist of abstract syntax trees

- no grammar rules for the variables

  imagine them being there
  or consider sets like Var to be a parameter of the definition

- multiple names for the same non-terminal

recap

- a set of lambda terms as strings called Λ
- relations $\equiv$, $=_\alpha$, $\to_\beta$, $\twoheadrightarrow_\beta$, $=_\beta$
- Curried functions
- fast-and-loose context-free grammars

homework for next Monday:

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

T-shirt

a lambda calculus evaluator

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

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

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

$$\mathsf{K} \equiv \lambda xy. x$$
$$\mathsf{S} \equiv \lambda xyz. xz(yz)$$
$$\mathsf{C} \equiv \lambda xyz. xzy$$

table of contents

contents