Introduction to Type Theory — Herman Geuvers

Lecture 3: Dependent Type Theory
OPLSS 2026

1 From propositional to predicate logic

The course progression is:

$$\begin{aligned} \text{propositional logic} &\leftrightarrow \lambda^{\to} \\ \text{second-order propositional logic} &\leftrightarrow \lambda 2 \\ \text{predicate logic} &\leftrightarrow \lambda P \\ \text{higher-order predicate logic} &\leftrightarrow \lambda C \\ \text{univalent foundations} &\leftrightarrow \text{homotopy type theory.} \end{aligned}$$

Predicate logic introduces two kinds of variables: first-order variables over a domain, and proof variables for assumptions. A predicate on D has type $D \to *$; a binary relation has type $D \to D \to *$.

Universal quantification is interpreted by dependent function space:

$$\forall x : D.A \quad \rightsquigarrow \quad \Pi x : D.A.$$

Introduction is lambda abstraction, elimination is application, and the beta rule $(\lambda x : D.M)t \to M[x := t]$ matches detour elimination in natural deduction.

Cut elimination in sequent calculus corresponds to detour elimination in natural deduction, just as beta reduction removes an introduction followed immediately by an elimination in the term calculus.

2 Dependent products

The main change from $\lambda^{\to}$ to λP is replacing a simple arrow $A \to B$ by a dependent product $\Pi x : A.B$. The codomain may now mention the input. If x is not free in B, then $\Pi x : A.B$ is just the ordinary function type $A \to B$.

Arrow is not primitive abstract syntax here: it is a special case of Π. When x does not occur in B, $\Pi x : A.B$ describes an ordinary function type.

For example, a proof of $\forall x : \mathbb{N}.x \geq 0$ is a term $p : \Pi x : \mathbb{N}.x \geq 0$ that maps every natural number n to a proof $p\,n : n \geq 0$. A term of $\Pi n : \mathbb{N}.Vec\ A\ n$ produces a vector whose length is the input number.

3 Rules of lambda-P

Pseudoterms are generated by

$$T ::= Var \mid * \mid \Box \mid (T\,T) \mid (\lambda x : T.T) \mid \Pi x : T.T.$$

Judgments have the form $\Gamma \vdash M : B$, and context well-formedness is checked simultaneously. The universe $*$ classifies types, and $\Box$ classifies $*$:

$$\vdash * : \Box.$$

We do not allow $* : *$.

$A : *$ means A is a type, while $A : \Box$ means A is a kind/universe-level object.

The conversion rule allows replacing a type by a beta-equal type:

$$\frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : s \qquad A =_\beta B}{\Gamma \vdash M : A}.$$

Excercize: from $P : Q(2+3)$ infer $P : Q(5)$.

4 Direct embedding of predicate logic

Minimal predicate logic with implication and universal quantification embeds directly into λP. Domains and formulas are represented as types:

$$A : *, \qquad P : A \to *, \qquad R : A \to A \to *.$$

Implication is represented by the nondependent arrow, and $\forall$ by Π.

The example

$$\forall z : A.(\forall x, y : A.Rxy) \to Rzz$$

is represented by the term

$$\lambda z : A.\lambda h : (\Pi x, y : A.Rxy).h\ z\ z : \Pi z : A.(\Pi x, y : A.Rxy) \to Rzz.$$

Rocq can create explicit proof terms of this kind. Tactics may build the term, but the trusted kernel checks the resulting lambda term.

The embedding is sound and complete at the level of provability: $\Gamma \vdash \varphi$ iff there exists a proof term M with the translated context proving the translated formula. The converse is subtler than for simply typed lambda-calculus because some λP terms do not correspond directly to ordinary predicate-logic proofs.

5 Logical frameworks

The second embedding style uses dependent type theory as a meta-language. In the LF approach of de Bruijn and Harper-Honsell-Plotkin, one declares a signature for the object logic. For minimal propositional logic:

$$prop : *, \qquad imp : prop \to prop \to prop, \qquad T : prop \to *.$$

Here $prop$ contains names of propositions, and $T\,p$ is the type of proofs of proposition p. The implication rules are declared as constants:

$$imp\_intr : \Pi p, q : prop.(T\,p \to T\,q) \to T(imp\ p\ q),$$

$$imp\_el : \Pi p, q : prop.T(imp\ p\ q) \to T\,p \to T\,q.$$

Propositions and their proofs are separated: $A \Rightarrow B : prop : *$, then $T(A \Rightarrow B) : *$, and finally $t : T(A \Rightarrow B)$ is a proof of $A \Rightarrow B$.

For predicate logic, $\forall : (A \to prop) \to prop$ is added, together with introduction and elimination constants:

$$\forall intr : \Pi P : A \to prop.(\Pi x : A.T(Px)) \to T(\forall P),$$

$$\forall elim : \Pi P : A \to prop.T(\forall P) \to \Pi x : A.T(Px).$$

LF can also encode untyped lambda-calculus using a type D of object-language terms, $app : D \to D \to D$, and $abs : (D \to D) \to D$, relying on the meta-language binder to represent object binding.

LF is strong enough to leak structural principles into object logics that should not have them. For example, ordinary LF has weakening, so representing a logic without weakening requires a refinement such as linear LF.

6 Type checking and kernels

Proof-assistant trust comes from separating a powerful proof engine from a small proof checker. Tactics build proof objects, and the kernel checks them against the rules. This is the De Bruijn criterion.

Type synthesis $TC_\Gamma(M)$ and context checking $Ok(\Gamma)$ are defined together. Soundness says

$$TC_\Gamma(M) = A \Rightarrow \Gamma \vdash M : A,$$

and completeness says

$$\Gamma \vdash M : A \Rightarrow TC_\Gamma(M) =_\beta A.$$

The hard cases are lambda abstraction and application. For application, the algorithm must recognize that the synthesized type of the function reduces to a product type and must decide definitional equality. Termination depends on strong normalization and Church-Rosser for well-typed terms.

Type checking pseudo-terms must terminate even on ill-typed inputs, but recursive calls and beta-equality checks make the argument delicate. Termination for the application case is obtained using soundness of type checking plus decidability of beta-equality on well-typed terms. Decidability of beta-equality on well-typed terms follows from strong normalization and Church-Rosser.

References

[1] N. G. de Bruijn. *A survey of the project Automath*. 1980.

[2] R. Harper, F. Honsell, and G. Plotkin. *A Framework for Defining Logics*. Journal of the ACM, 1993.