Introduction to Type Theory — Herman Geuvers

Lecture 1b: Simple Type Theory
OPLSS 2026

1 Simple types

Simple type theory, written $\lambda^{\to}$, adds arrow types to lambda terms:

$$Typ ::= TVar \mid (Typ \to Typ).$$

Arrows associate to the right, so $A \to B \to C$ means $A \to (B \to C)$. Contexts record free-variable declarations $x_1 : A_1, \ldots, x_n : A_n$.

In the Church-style presentation, abstractions carry type annotations:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \qquad \frac{\Gamma, x : A \vdash P : B}{\Gamma \vdash \lambda x : A.P : A \to B}.$$

This lets one read derivation rules bottom-up as algorithms for type checking, type synthesis, and type inhabitation.

Type checking and synthesis are equivalent for this system. To check $MN : A$, synthesize a type B for N, then check that M has type $B \to A$.

2 Church and Curry typing

The Curry-style system assigns types to unannotated lambda terms:

$$\frac{\Gamma, x : A \vdash P : B}{\Gamma \vdash \lambda x.P : A \to B}.$$

Church terms have more information and therefore unique types. Curry terms may have many types, but a principal type can be computed by unification in the simply typed setting. This distinction matches two use cases: programming languages often prefer Curry-style inference, while proof assistants benefit from explicit structure, especially once dependent types are introduced.

Milner's slogan "well typed programs don't go wrong" gives the operational intuition behind type soundness; the original technical development appears in Milner's 1978 paper on ML-style polymorphism.

3 Proofs as typed terms

The formulas-as-types correspondence identifies $\lambda^{\to}$ terms with derivations in minimal propositional logic restricted to implication. A judgment

$$x_1 : B_1, \ldots, x_n : B_n \vdash M : A$$

can be read as: M is a proof of A from assumptions $B_1, \ldots, B_n$.

The proof of

$$(A \to B \to C) \to (A \to B) \to A \to C$$

corresponds to the term

$$\lambda x : A \to B \to C . \lambda y : A \to B . \lambda z : A . \, x\, z\, (y\, z).$$

The Fitch-style derivation exposes the same dependency structure linearly: assume x, then y, then z, derive $xz : B \to C$ and $yz : B$, apply, and abstract over z, y, and x.

4 Computation as detour elimination

In natural deduction, a detour is an introduction rule immediately followed by the matching elimination rule. In $\lambda^{\to}$, this is exactly a β-redex:

$$(\lambda x : A . M) P \to_\beta M[x := P].$$

Thus computation corresponds to proof normalization. A proof of $A \to A \to B, (A \to B) \to A \vdash B$ contains a detour whose term-level witness contains the redex

$$(\lambda x : A . p\, x\, x)(q(\lambda x : A . p\, x\, x)).$$

η reduction was introduced as the rule $\lambda x . M x \to_\eta M$, provided x is not free in M. This says that a function which only applies M to its argument is extensionally the same as M itself.

5 Algorithms and metatheory

For simply typed lambda-calculus, type checking, synthesis, and inhabitation are decidable in both Curry and Church styles. Type-synthesis and Type-checking are equivalent problems, we need to solve one to solve the other. For example, if we want to check weather a term $M\ N$ has type A, we need to know that M has type $B \to A$ and N has type B. But we do not know what B is, leading us to a type synthesis problem!

Inhabitation is already nontrivial: for $\lambda^{\to}$ it is PSPACE-complete, and for most extensions it becomes undecidable because it corresponds to provability in a richer logic.

The main metatheoretic properties are uniqueness of types for Church terms, subject reduction, and strong normalization. Subject reduction says reduction preserves types: if $\Gamma \vdash M : A$ and $M \to_{\beta\eta} N$, then $\Gamma \vdash N : A$. Strong normalization says every well-typed term terminates under all $\beta\eta$-reduction sequences.

Products and sums extend the correspondence to conjunction and disjunction. For products, $\langle P, Q\rangle : A \times B$ introduces a pair and projections eliminate it, with $\pi_1\langle P, Q\rangle \to P$ and $\pi_2\langle P, Q\rangle \to Q$.

6 Polymorphic types and the $\forall$ rules

In the simply typed calculus a function cannot be reused at different types:

$$\lambda x{:}\alpha.\, x : \alpha \to \alpha \qquad \lambda x{:}\beta.\, x : \beta \to \beta$$

(cannot re-use one function). Polymorphism abstracts over the type:

$$\forall\alpha.\, AM.\, \forall\alpha.\, \alpha \to \alpha$$

$$\forall\alpha.\, \forall\beta.\, \alpha \to \beta \to \alpha$$

6.1 Type grammar

$$\text{Typ} \;::=\; \text{Tvar} \;\mid\; \text{Typ} \to \text{Typ} \;\mid\; \forall\alpha.\, \text{Typ}$$

6.2 Generalization and instantiation

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \lambda\alpha.\, M : \forall\alpha.\, A}\ (\alpha \notin (\Gamma)) \qquad\qquad \frac{\Gamma \vdash M : \forall\alpha.\, A}{\Gamma \vdash M\, B : A[\alpha := B]}$$

6.3 Worked example: the polymorphic combinator Z

$$\begin{aligned} Z &: \forall\alpha.\, \alpha \to \alpha \\ Z\,(\alpha \to \alpha) &: (\alpha \to \alpha) \to (\alpha \to \alpha) \\ Z\,\alpha &: \alpha \to \alpha \\ Z\,(\alpha \to \alpha)\,(Z\,\alpha) &: \alpha \to \alpha \\ \lambda\alpha.\, Z\,(\alpha \to \alpha)\,(Z\,\alpha) &: \forall\alpha.\, \alpha \to \alpha \end{aligned}$$

6.4 Bottom and top

$$\bot := \forall\alpha.\, \alpha \qquad\qquad \top := \forall\alpha.\, \alpha \to \alpha$$

7 Girard's theorem and the identity at $\bot$

Since $\bot = \forall\alpha.\,\alpha$ is the universal quantification rule, it gives an elimination principle:

$$\frac{M : \bot}{M : A} \qquad\qquad \frac{M : \top}{M : A \to A}$$

Girard's theorem. Only functions provably total in second-order arithmetic can be defined in $\lambda 2$.

7.1 Deriving the polymorphic identity at $\bot$

Reading the flag-style derivation (assume $x : \bot$, then α):

$$\begin{array}{l} x : \bot \\ \quad \alpha : * \\ \quad x\,((\alpha \to \alpha)) : \alpha \to \alpha \\ \quad x : \alpha \quad\;\; : \alpha \\ \quad x\,(\alpha \to \alpha)\,(x : \alpha) : \alpha \\ \quad \lambda\alpha.\,\big(x\,(\alpha \to \alpha)\,(x : \alpha)\big) : \bot \end{array}$$

so

$$\lambda x{:}\bot.\,\lambda\alpha.\,x\,(\alpha \to \alpha)\,(x\,\alpha) \;:\; \bot \to \bot \qquad \big(\equiv \lambda x.\,x\big)$$

7.2 Church versus Curry style

$$\text{Church } \lambda 2 \quad \overset{\|\ \text{(erasure)}}{\underset{\exists}{\rightleftarrows}} \quad \text{Curry } \lambda 2$$

The forward map is type erasure $|\cdot|$; the backward direction (type reconstruction) is an existence question.

8 Encoding connectives and data

8.1 Disjunction elimination, encoded

The $\vee$-elimination rule

$$\dfrac{A \vee B \qquad \begin{array}{c}[A]\\ \vdots \\ C\end{array} \qquad \begin{array}{c}[B]\\ \vdots \\ C\end{array}}{C}$$

is captured by reading the eliminator's type. With $C := \alpha$:

$$\forall\alpha.\,(A \to \alpha) \to (B \to \alpha) \to \alpha$$

(the elimination rule is what is being said by the type).

A similar reading recovers, e.g.,

$$\forall\alpha.\,(A \to B \to \alpha) \to \alpha \qquad \frac{(A \to B \to A) \to A \qquad \dfrac{[A]}{B \to A} \quad A \to B \to A}{A}$$

8.2 Closed types in the polymorphic lambda-calculus

$$\mathrm{Nat} := \forall\alpha.\,\alpha \to (\alpha \to \alpha) \to \alpha$$

With α a type, $x : \alpha$, and $f : \alpha \to \alpha$, the body $f(f(f\,x)) : \alpha$ gives the numeral 3:

$$\frac{f(f(f\,x)) : \alpha}{\lambda\alpha.\,\lambda x{:}\alpha.\,\lambda f{:}\alpha \to \alpha.\,f\,(f\,(f\,x)) : \mathrm{Nat}}$$

References

[1] R. Milner. *A Theory of Type Polymorphism in Programming.* Journal of Computer and System Sciences, 1978.