FLOLAC 2026

λ-Calculus

Simple Types and their Extensions

陳亮廷 Chen, Liang-Ting

Formosan Summer School on Logic, Language, and Computation 2026

Institute of Information Science
Academia Sinica

Recap and plan

In the previous lectures, we studied untyped λ-calculus:

- β-reduction and evaluation strategies;
- encodings of data and recursion;
- confluence and normalisation.

Today we add types and study:

1. the simply typed λ-calculus;
2. typed programming with Church encodings;
3. type safety: preservation and progress;
4. if time permits, extensions with general recursion and primitive data.

Simply Typed λ-Calculus: Introduction

Untyped λ-calculus is expressive and computationally powerful, but it is difficult to read programs from their syntax alone.

A function can be applied to any term: a Boolean encoding, a Church numeral, another function, or a divergent term. As a programming language, the untyped calculus does not make the programmer's intention explicit.

We introduce a typing judgement

$$\Gamma \vdash t : A,$$

meaning that the term t has type A under the assumptions in the context Γ.

Simply Typed λ-Calculus: Statics

Assume that $\mathbb{V}$ is a set of type variables, distinct from term variables. We suppress this distinction when it is clear from context.

Definition 1

The judgement $A : \mathtt{Type}$ is defined inductively as follows.

$$\frac{}{X : \mathtt{Type}} \text{ if } X \in \mathbb{V} \qquad \frac{A : \mathtt{Type} \qquad B : \mathtt{Type}}{A \to B : \mathtt{Type}}$$

We say that A is a type if $A : \mathtt{Type}$ is derivable. The type $A \to B$ is the type of functions from A to B.

Function types are higher-order because

1. functions can be arguments of other functions;
2. functions can be returned as results.

For example,

$(A_1 \to A_2) \to B$ is the type of functions whose argument has type $A_1 \to A_2$;
$A_1 \to (A_2 \to B)$ is the type of functions returning a function of type $A_2 \to B$.

Convention

Function types associate to the right:

$$A_1 \to A_2 \to \cdots \to A_n := A_1 \to (A_2 \to (\cdots \to (A_{n-1} \to A_n) \cdots)).$$

Definition 2

A *typing context* Γ is a sequence

$$\Gamma \equiv x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n$$

of distinct variables x_i with types A_i.

Definition 3

The membership judgement $\Gamma \ni (x : A)$ is defined inductively:

$$\frac{}{\Gamma, x : A \ni x : A} \qquad \frac{\Gamma \ni x : A}{\Gamma, y : B \ni x : A}$$

We say that x of type A occurs in Γ if $\Gamma \ni (x : A)$ is derivable.

Work out the following derivations.

1. Derive
$$X \to Y \to X : \mathsf{Type}.$$
2. Let
$$\Gamma \equiv x : A,\ y : B,\ z : A \to B.$$
Which of the following membership judgements are derivable?
$$\Gamma \ni x : A, \qquad \Gamma \ni y : A, \qquad \Gamma \ni z : A \to B.$$

The implicit typing system for simply typed λ-calculus is defined by the following rules.

$$\frac{}{\Gamma \vdash_i x : A}\ \text{(var)} \quad \text{if } \Gamma \ni (x : A)$$

$$\frac{\Gamma, x : A \vdash_i t : B}{\Gamma \vdash_i \lambda x.\, t : A \to B}\ \text{(abs)}$$

$$\frac{\Gamma \vdash_i t : A \to B \qquad \Gamma \vdash_i u : A}{\Gamma \vdash_i t\ u : B}\ \text{(app)}$$

We say that t is a closed typed term if $\vdash_i t : A$ is derivable for some type A.

A typing system is *syntax-directed* if it has exactly one typing rule for each term constructor.

Since this typing system is syntax-directed, every typing derivation can be inverted.

Lemma 4 (Typing inversion)

Suppose that $\Gamma \vdash_i t : A$ *is derivable.*

$t \equiv x$ *implies* $\Gamma \ni (x : A)$.

$t \equiv \lambda x.\, t'$ *implies* $A = B \to C$ *and* $\Gamma, x : B \vdash_i t' : C$ *for some types* B *and* C.

$t \equiv u\ v$ *implies there is a type* B *such that* $\Gamma \vdash_i u : B \to A$ *and* $\Gamma \vdash_i v : B$.

For any types A and B, the judgement

$$\vdash_i \lambda x\, y.\, x : A \to B \to A$$

has the following derivation.

$$\dfrac{\dfrac{\dfrac{}{x : A, y : B \vdash_i x : A}\,(\text{var})}{x : A \vdash_i \lambda y.\, x : B \to A}\,(\text{abs})}{\vdash_i \lambda x\, y.\, x : A \to B \to A}\,(\text{abs})$$

Therefore, $\lambda x\, y.\, x$ is a program of type $A \to B \to A$.

Derive the typing judgement

$$\vdash_i \lambda f\, g\, x.\, f\, x\, (g\, x) : (A \to B \to C) \to (A \to B) \to A \to C$$

for arbitrary types A, B, and C.

Hint: first derive the following judgements under the context $f : A \to B \to C,\ g : A \to B,\ x : A$:

$$f\, x : B \to C, \qquad g\, x : B.$$

When we try to assign a type to a term, the typing rules generate equality constraints between types. We call these constraints type equations.

For example, to type an application $t\ u$, the function part must have a function type:

$$\Gamma \vdash_i t : A \to B, \qquad \Gamma \vdash_i u : A.$$

If t has already been assigned type C, then we must solve

$$C = A \to B.$$

In simply typed λ-calculus, types are finite trees built from type variables and $\to$. Thus an equation such as

$$A = A \to B$$

has no solution among simple types.

Exercise: inversion and non-typability

Use typing inversion to decide whether the following terms are typable.

1. $\lambda x.\, x\ x$
2. $\lambda f\, x.\, f\ x$
3. $\lambda x\, y.\, x\ (y\ x)$

For typable terms, give a possible type. For non-typable terms, identify the unsolvable type equation that would be required.

Programming in Simply Typed λ-Calculus

For any type A, define the Church-numeral type at A by

$$\mathsf{nat}_A := (A \to A) \to A \to A.$$

Church numerals

$$\mathbf{c}_n := \lambda f\, x.\, f^n x$$
$$\vdash_i \ \mathbf{c}_n : \mathsf{nat}_A$$

Successor

$$\mathsf{suc} := \lambda n\, f\, x.\, f\,(n\ f\ x)$$
$$\vdash_i \ \mathsf{suc} : \mathsf{nat}_A \to \mathsf{nat}_A$$

Addition

$$\mathtt{add} := \lambda n\, m\, f\, x.\, m\ f\ (n\ f\ x)$$
$$\vdash_i\ \mathtt{add} : \mathtt{nat}_A \to \mathtt{nat}_A \to \mathtt{nat}_A$$

Multiplication

$$\mathtt{mul} := \lambda n\, m\, f\, x.\, m\ (n\ f)\ x$$
$$\vdash_i\ \mathtt{mul} : \mathtt{nat}_A \to \mathtt{nat}_A \to \mathtt{nat}_A$$

Zero test

$$\mathtt{ifz} := \lambda n\, x\, y.\, n\ (\lambda z.\, y)\ x$$
$$\vdash_i\ \mathtt{ifz} : \mathtt{nat}_A \to A \to A \to A$$

Work in the simply typed λ-calculus.

1. Show that
$$\vdash_i \mathtt{suc} : \mathtt{nat}_A \to \mathtt{nat}_A.$$
2. Show that
$$\vdash_i \mathtt{ifz} : \mathtt{nat}_A \to A \to A \to A.$$
3. Check that
$$\mathtt{ifz}\ \mathbf{c}_0\ t\ u \longrightarrow\!\!\!\!\!\rightarrow_\beta t, \qquad \mathtt{ifz}\ \mathbf{c}_{n+1}\ t\ u \longrightarrow\!\!\!\!\!\rightarrow_\beta u.$$

For any type A, define

$$\texttt{bool}_A := A \to A \to A.$$

Boolean values

$$\texttt{true} := \lambda x\, y.\, x \qquad \text{and} \qquad \texttt{false} := \lambda x\, y.\, y.$$

Conditional

$$\texttt{cond} := \lambda b\, x\, y.\, b\, x\, y$$
$$\vdash_i \ \texttt{cond} : \texttt{bool}_A \to A \to A \to A$$

Exercise: typed Boolean operations

Define conjunction, disjunction, and negation in simply typed λ-calculus.

1. Give terms for
$$\mathtt{and}, \qquad \mathtt{or}, \qquad \mathtt{not}.$$
2. Check that the following judgements are derivable:
$$\vdash_i \mathtt{and} : \mathtt{bool}_A \to \mathtt{bool}_A \to \mathtt{bool}_A,$$
$$\vdash_i \mathtt{or} : \mathtt{bool}_A \to \mathtt{bool}_A \to \mathtt{bool}_A, \qquad \vdash_i \mathtt{not} : \mathtt{bool}_A \to \mathtt{bool}_A.$$

Type Safety

"Well-typed programs cannot go wrong."

– Milner, 1978

Preservation If $\Gamma \vdash_i t : A$ and $t \longrightarrow_\beta u$, then $\Gamma \vdash_i u : A$.

Progress If $\Gamma \vdash_i t : A$, then either t is in normal form or there exists u such that $t \longrightarrow_\beta u$.

Preservation says that reduction does not change types. Progress says that a well-typed term is never stuck except at a normal form.

The converse of preservation does not hold: a reduct may be typable even when the original term is not.

Lemma 5 (Typability of subterms)

If $\Gamma \vdash_i t : A$ is derivable, then every subterm of t is typable under some context.

Let

$$\mathbf{K}_1 := \lambda x\, y.\, x, \qquad \mathbf{I} := \lambda x.\, x, \qquad \Omega := (\lambda x.\, x\, x)\, (\lambda x.\, x\, x).$$

Then

$$\mathbf{K}_1\ \mathbf{I}\ \Omega \longrightarrow\!\!\!\!\!\rightarrow_\beta \mathbf{I}.$$

The reduct $\mathbf{I}$ is typable, but Ω is not typable, so $\mathbf{K}_1\ \mathbf{I}\ \Omega$ is not typable.

We use two standard lemmas.

Weakening If $\Gamma \vdash_i t : A$ and $x \notin \mathrm{dom}(\Gamma)$, then $\Gamma, x : B \vdash_i t : A$.

Substitution If $\Gamma, x : A \vdash_i t : B$ and $\Gamma \vdash_i u : A$, then $\Gamma \vdash_i t[u/x] : B$.

Theorem 6 (Preservation)

If $\Gamma \vdash_i t : A$ and $t \longrightarrow_\beta u$, then $\Gamma \vdash_i u : A$.

Proof sketch.

By induction on the derivations of $\Gamma \vdash_i t : A$ and $t \longrightarrow_\beta u$. The only interesting case is the β-redex $(\lambda x.\, t')\ u'$, where the substitution lemma is needed. □

Exercise: preservation for the β-case

Fill in the main case of the preservation proof.

Suppose

$$\Gamma \vdash_i (\lambda x.\, t)\ u : B \qquad \text{and} \qquad (\lambda x.\, t)\ u \longrightarrow_\beta t[u/x].$$

1. Use inversion to obtain a type A such that

$$\Gamma \vdash_i \lambda x.\, t : A \to B, \qquad \Gamma \vdash_i u : A.$$

2. Use inversion again to obtain

$$\Gamma, x : A \vdash_i t : B.$$

3. Apply substitution to conclude

$$\Gamma \vdash_i t[u/x] : B.$$

To prove progress constructively, we use a syntactic characterisation of normal forms.

$$\frac{}{\texttt{Neutral}\ x}$$

$$\frac{\texttt{Neutral}\ t \qquad \texttt{Normal}\ u}{\texttt{Neutral}\ (t\ u)}$$

$$\frac{\texttt{Neutral}\ t}{\texttt{Normal}\ t}$$

$$\frac{\texttt{Normal}\ u}{\texttt{Normal}\ (\lambda x.\, u)}$$

Neutral terms are variables applied to normal arguments. Normal terms are either neutral terms or abstractions whose bodies are normal.

Decide which of the following terms are neutral, normal, or reducible.

1. x
2. $x\ ((\lambda y.\, y)\ z)$
3. $\lambda x.\, x\ y$
4. $(\lambda x.\, x)\ y$
5. $x\ (\lambda y.\, y)$

For each normal term, give a derivation of `Normal` t or `Neutral` t.

Lemma 7 (Normal-form characterisation)

A term t has no β-reduction if and only if `Normal` t *is derivable.*

This lemma says that the inductive judgements `Normal` and `Neutral` exactly capture the ordinary notion of having no β-redex.

We will use this characterisation in the proof of progress, but we will not prove the lemma here.

Theorem 8 (Progress)

If $\Gamma \vdash_i t : A$*, then* $\texttt{Normal}\ t$ *or there exists* u *such that* $t \longrightarrow_\beta u$*.*

Proof sketch.

By induction on the derivation of $\Gamma \vdash_i t : A$.

The variable and abstraction cases produce normal forms. The application case is the main case: if the function part reduces, the whole application reduces; if the function part is an abstraction, there is a β-redex; otherwise the application is neutral when its argument is normal. □

Fill in the application case of the progress proof.

Suppose

$$\Gamma \vdash_i t\ u : B.$$

By inversion, for some type A,

$$\Gamma \vdash_i t : A \to B, \qquad \Gamma \vdash_i u : A.$$

Use the induction hypotheses for t and u to decide whether $t\ u$ is normal or reducible. Pay special attention to the case where t is an abstraction.

Type safety: summary

Preservation and progress are proved separately, but they work together.

Preservation Each reduction step keeps the same type.

Progress A well-typed term is either already normal or can take a reduction step.

Hence evaluation of a well-typed term cannot get stuck because of a type error. It either continues reducing while preserving its type, or it reaches a normal form.

Definition 9

A term t is *strongly normalising*, written $t \Downarrow$, if every reduction sequence starting from t terminates.

Theorem 10 (Strong normalisation)

Every typable term t with $\Gamma \vdash_i t : A$ is strongly normalising.

This theorem is stronger than type safety. Type safety says that reduction does not get stuck; strong normalisation says that, in the simply typed λ-calculus, reduction cannot continue forever.

We state this result without proof.

Optional Extensions

The self-applicative term

$$\lambda x.\, x\ x$$

is not typable in simply typed λ-calculus: if $x : A$, then using x as a function applied to itself would require $A = A \to B$ for some type B.

Hence the untyped Y combinator is not typable.

We can add general recursion explicitly by extending the term grammar with

$$\mathtt{fix}\, f.\, t.$$

$$\frac{\Gamma, f : A \vdash_i t : A}{\Gamma \vdash_i \mathtt{fix}\, f.\, t : A}$$

Reduction is extended with the rule

$$\mathtt{fix}\, f.t \longrightarrow_\beta t[\mathtt{fix}\, f.t/f].$$

Non-terminating well-typed terms are now easy to define. For any type A,

$$\vdash_i \mathtt{fix}\, x.x : A.$$

Its reduction unfolds forever:

$$\begin{aligned}\mathtt{fix}\, x.x &\longrightarrow_\beta x[\mathtt{fix}\, x.x/x] \\ &\equiv \mathtt{fix}\, x.x \\ &\longrightarrow_\beta \cdots\end{aligned}$$

Adding general recursion therefore destroys strong normalisation.

Exercise: fixed points and type safety

Work in the calculus extended with `fix`.

1. Derive
$$\vdash_i \texttt{fix}\, x.\, x : A.$$
2. Evaluate the first three reduction steps of
$$\texttt{fix}\, x.\, x.$$
3. Does preservation still hold for the `fix` reduction rule? Explain which typing rule is used.
4. Does strong normalisation still hold?

Church numerals have many possible types $\mathtt{nat}_A$. Instead, we can extend the calculus with a single primitive type of natural numbers.

Add terms:

- $\mathtt{zero}$;
- $\mathtt{suc}(t)$, if t is a term;
- $\mathtt{ifz}(t; x.\, u; v)$, if t, u, and v are terms.

Add typing rules:

$$\frac{}{\Gamma \vdash_i \mathtt{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash_i t : \mathbb{N}}{\Gamma \vdash_i \mathtt{suc}(t) : \mathbb{N}} \qquad \frac{\Gamma \vdash_i v : \mathbb{N} \qquad \Gamma \vdash_i t : A \qquad \Gamma, x : \mathbb{N} \vdash_i u : A}{\Gamma \vdash_i \mathtt{ifz}(t; x.\, u; v) : A}$$

The third rule is pattern matching on natural numbers.

Reduction for natural numbers is extended with two rules:

$$\mathtt{ifz}(t; x.\, u; \mathtt{zero}) \longrightarrow_\beta t$$
$$\mathtt{ifz}(t; x.\, u; \mathtt{suc}(n)) \longrightarrow_\beta u[n/x].$$

Intuitively,

$$\mathtt{ifz}(t; x.\, u; v)$$

means: inspect v; if it is zero, return t; if it is $\mathtt{suc}(n)$, bind n to x and return u.

Define predecessor as a program

$$\texttt{pred} : \mathbb{N} \to \mathbb{N}.$$

Hint:

$$\texttt{pred} := \lambda n.\, \texttt{ifz}(\texttt{zero}; x.\, x; n).$$

Evaluate the following terms to their normal forms.

1. $\texttt{pred}\ \texttt{zero}$
2. $\texttt{pred}\ (\texttt{suc}(\texttt{suc}(\texttt{suc}(\texttt{zero}))))$

Exercise: primitive Boolean values

Extend $\Lambda_{\mathtt{fix},\mathbb{N}}(V)$ further with a primitive Boolean type.

1. What term constructors should be added?
2. What typing rules should be added?
3. How should reduction be extended?
4. Define conjunction, disjunction, and negation in this extension.

Aim for a design with constructors

$$\mathtt{true}, \qquad \mathtt{false},$$

and an eliminator analogous to pattern matching.

These exercises are for peer discussion.

1. Complete the application case of the Progress Theorem.
2. Show that if a term is in normal form, then `Normal` t is derivable.
3. If time permits, extend the calculus with product types $A \times B$:
   - 3.1 add terms for pairs;
   - 3.2 add a pattern-matching eliminator;
   - 3.3 write the typing rules;
   - 3.4 write the reduction rule.