FLOLAC 2026

λ-Calculus

Parametric Polymorphism

陳亮廷 Chen, Liang-Ting

Formosan Summer School on Logic, Language, and Computation 2026

Institute of Information Science
Academia Sinica

Where we are

In the simply typed λ-calculus, a term receives one type at a time. For example,

$$\lambda x.\, x : A \to A$$

for each chosen type A.

This lecture asks what happens if a single program may be used uniformly at every type.

Simply typed λ-calculus	one type at a time
Polymorphic λ-calculus	one term for all types

The technical object is System F: simply typed λ-calculus extended with universal quantification over types.

Polymorphic λ-Calculus: Static

Many programs do not inspect the concrete type of their inputs.

Identity

$$\texttt{id} : \forall X.\, X \to X$$

should work at every type X.

First projection

$$\texttt{proj}_1 : \forall X.\, \forall Y.\, X \to Y \to X$$

should ignore the second argument uniformly.

Church numerals

$$\texttt{nat} := \forall X.\, (X \to X) \to X \to X$$

should iterate any endofunction on any carrier type.

System F adds two new constructs.

	Introduction	Elimination
term abstraction	$\lambda x : A.\, t$	$t\, u$
type abstraction	$\lambda X.\, t$	$t\, A$

A term of type

$$\forall X.\, A$$

is a program that can be instantiated at any type B:

$$t\, B : A[B/X].$$

Fix a countably infinite set $\mathbb{V}$ of type variables. A type context Δ is a finite sequence of distinct type variables.

Definition 1

The judgement $\Delta \vdash A : \mathsf{Type}$ is defined inductively as follows.

$$\frac{X \in \Delta}{\Delta \vdash X : \mathsf{Type}}\ \textsc{TVar} \qquad \frac{\Delta \vdash A : \mathsf{Type} \qquad \Delta \vdash B : \mathsf{Type}}{\Delta \vdash A \to B : \mathsf{Type}}\ \textsc{TFun}$$

$$\frac{\Delta, X \vdash A : \mathsf{Type}}{\Delta \vdash \forall X.\, A : \mathsf{Type}}\ \textsc{TAll}$$

Free type variables are computed by

$$\begin{aligned}\mathbf{FV}(X) &= \{X\},\\ \mathbf{FV}(A \to B) &= \mathbf{FV}(A) \cup \mathbf{FV}(B),\\ \mathbf{FV}(\forall X.\, A) &= \mathbf{FV}(A) - \{X\}.\end{aligned}$$

The type variable X is bound in $\forall X.\, A$.

Example 2

$$(\forall Y.\, X \to Y)[B/X] = \forall Y.\, B \to Y$$

provided that $Y \notin \mathbf{FV}(B)$.

Definition 3

System F extends untyped terms with type abstraction and type application:

variable	$x \in \Lambda_\forall(V, \mathbb{V})$ if $x \in V$;
application	$t \mathbin{@} u \in \Lambda_\forall(V, \mathbb{V})$ if $t, u \in \Lambda_\forall(V, \mathbb{V})$;
abstraction	$\lambda x : A.\, t \in \Lambda_\forall(V, \mathbb{V})$ if $x \in V$, A is a type, and $t \in \Lambda_\forall(V, \mathbb{V})$;
type abstraction	$\lambda X.\, t \in \Lambda_\forall(V, \mathbb{V})$ if $X \in \mathbb{V}$ and $t \in \Lambda_\forall(V, \mathbb{V})$;
type application	$t\ A \in \Lambda_\forall(V, \mathbb{V})$ if $t \in \Lambda_\forall(V, \mathbb{V})$ and A is a type.

Type abstraction abstracts over a type variable:

$$\lambda X.\, t.$$

System F has two contexts.

Type context

$$\Delta ::= \cdot \mid \Delta, X$$

records available type variables.

Term context

$$\Gamma ::= \cdot \mid \Gamma, x : A$$

records available term variables and their types.

The main judgements are

$$\Delta \vdash A : \mathsf{Type} \qquad \text{and} \qquad \Delta; \Gamma \vdash t : A.$$

The judgement $\vdash t : A$ abbreviates $\cdot; \cdot \vdash t : A$.

The typing rules for variables, abstraction, and application are inherited from simply typed λ-calculus with the additional type context Δ.

$$\frac{}{\Delta; \Gamma \vdash x : A} \text{ if } \Gamma \ni x : A$$

$$\frac{\Delta \vdash A : \mathsf{Type} \qquad \Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda(x : A).\, t : A \to B}$$

$$\frac{\Delta; \Gamma \vdash t : A \to B \qquad \Delta; \Gamma \vdash u : A}{\Delta; \Gamma \vdash t\ u : B}$$

The new rules introduce and eliminate universal types.

$$\frac{\Delta, X; \Gamma \vdash t : A}{\Delta; \Gamma \vdash \lambda X.\, t : \forall X.\, A}\ \forall\text{-intro}$$

$$\frac{\Delta; \Gamma \vdash t : \forall X.\, A \qquad \Delta \vdash B : \mathsf{Type}}{\Delta; \Gamma \vdash t\ B : A[B/X]}\ \forall\text{-elim}$$

The side condition in $\forall$-introduction is carried by the type context: any occurrence of X in A is bound by the introduced quantifier.

The polymorphic identity function is $\mathtt{id} := \lambda X.\, \lambda(x : X).\, x$.

Its type is derivable as follows.

$$\dfrac{\dfrac{\dfrac{}{X \vdash X : \mathtt{Type}} \quad \dfrac{}{X; x : X \vdash x : X}}{X; \cdot \vdash \lambda(x : X).\, x : X \to X}}{\cdot; \cdot \vdash \lambda X.\, \lambda(x : X).\, x : \forall X.\, X \to X}$$

Instantiation at a type A gives $\mathtt{id}\ A : A \to A$.

Exercise

Reduce the following term:

$$\mathtt{id}\ (A \to A)\ (\lambda(x : A).\, x).$$

What is its type?

Derive the following judgements.

1. $$\vdash \lambda X.\, \lambda Y.\, \lambda (x : X).\, \lambda (y : Y).\, x : \forall X.\, \forall Y.\, X \to Y \to X.$$
2. $$\vdash \lambda X.\, \lambda (f : X \to X).\, \lambda (x : X).\, f\ (f\ x) : \forall X.\, (X \to X) \to X \to X.$$
3. Explain why a term of type $\forall X.\, X \to X$ cannot use any operation specific to a particular type.

Polymorphic λ-Calculus: Dynamics and Programming

System F has the ordinary term-level β-rule and a type-level β-rule.

$$(\lambda(x : A).\, t)\ u \longrightarrow_\beta t[u/x],$$
$$(\lambda X.\, t)\ A \longrightarrow_\beta t[A/X].$$

Example:

$$\begin{aligned}(\lambda X.\, \lambda(x : X).\, x)\ A\ u &\longrightarrow_\beta (\lambda(x : X).\, x)[A/X]\ u \\ &\equiv (\lambda(x : A).\, x)\ u \\ &\longrightarrow_\beta u.\end{aligned}$$

The same structural proof pattern as in simply typed λ-calculus extends to System F.

Theorem 4 (Type safety)

If $\Delta; \Gamma \vdash t : A$ and $t \longrightarrow_\beta u$, then $\Delta; \Gamma \vdash u : A$.

Theorem 5 (Progress)

If $\cdot; \cdot \vdash t : A$, then either t is normal or there exists u such that $t \longrightarrow_\beta u$.

System F also enjoys strong normalisation: every well-typed reduction sequence terminates. We state this theorem without proof.

A binary sum can be encoded by asking the sum value how to eliminate itself.

Definition 6

$$A + B := \forall X.\, (A \to X) \to (B \to X) \to X.$$

Read this as:

To use an $A + B$, give a result type X, a case for A, and a case for B.

The injections are

$$\texttt{left}_{A,B} := \lambda(x : A).\, \lambda X.\, \lambda(f : A \to X).\, \lambda(g : B \to X).\, f\ x,$$
$$\texttt{right}_{A,B} := \lambda(y : B).\, \lambda X.\, \lambda(f : A \to X).\, \lambda(g : B \to X).\, g\ y.$$

Exercise

Define

$$\texttt{either} : \forall X.\, (A \to X) \to (B \to X) \to A + B \to X.$$

The polymorphic type

$$\bot := \forall X.\, X$$

behaves as an empty type: a closed term of this type would have to produce an element of every type.

Exercise

Think informally about what a closed term of type $\forall X.\, X$ would have to do.

- It must work after X is instantiated with any type.
- It receives no term argument from which an X could be obtained.
- So where could a value of the chosen type come from?

We will return to a parametricity argument for non-existence later.

A binary product can be encoded by asking the pair how to eliminate itself.

Definition 7

$$A \times B := \forall X.\, (A \to B \to X) \to X.$$

Pairing is

$$\langle -, - \rangle_{A,B} := \lambda(x : A).\, \lambda(y : B).\, \lambda X.\, \lambda(f : A \to B \to X).\, f\ x\ y.$$

Exercise

Define

$$\texttt{split} : A \times B \to \forall X.\, (A \to B \to X) \to X.$$

Then use it to define projections

$$\texttt{proj}_1 : A \times B \to A \qquad \text{and} \qquad \texttt{proj}_2 : A \times B \to B.$$

Church numerals become a single closed polymorphic type:

$$\mathsf{nat} := \forall X.\, (X \to X) \to X \to X.$$

Numerals

$$\mathbf{c}_n := \lambda X.\, \lambda (f : X \to X).\, \lambda (x : X).\, f^n\ x.$$

Successor

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

Fold

$$\mathsf{fold}_{\mathsf{nat}} := \lambda X.\, \lambda (f : X \to X).\, \lambda (x : X).\, \lambda (n : \mathsf{nat}).\, n\ X\ f\ x.$$

Addition and multiplication can be written directly:

$$\texttt{add} := \lambda(m : \texttt{nat}).\, \lambda(n : \texttt{nat}).\, \lambda X.\, \lambda(f : X \to X).\, \lambda(x : X).\, m\ X\ f\ (n\ X\ f\ x),$$
$$\texttt{mul} := \lambda(m : \texttt{nat}).\, \lambda(n : \texttt{nat}).\, \lambda X.\, \lambda(f : X \to X).\, \lambda(x : X).\, m\ X\ (n\ X\ f)\ x.$$

Exercise

Rewrite `add` and `mul` using $\texttt{fold}_{\texttt{nat}}$.

Lists over a type A are encoded by their fold operator.

Definition 8

$$\texttt{list}(A) := \forall X.\, X \to (A \to X \to X) \to X.$$

Constructors:

$$\texttt{nil}_A := \lambda X.\, \lambda (z : X).\, \lambda (c : A \to X \to X).\, z,$$
$$\texttt{cons}_A := \lambda (x : A).\, \lambda (xs : \texttt{list}(A)).\, \lambda X.\, \lambda (z : X).\, \lambda (c : A \to X \to X).\, c\ x\ (xs\ X\ z\ c).$$

Exercise

Define

$$\texttt{map} : (A \to B) \to \texttt{list}(A) \to \texttt{list}(B).$$

Reasoning with Types

Types rule out bad programs, but polymorphic types can also rule out many seemingly possible behaviours.

In fact, types can be used to tell which functions are definable, and which equations a term should satisfy at a given type.

For example, what can a closed term of each type do?

1. $\forall X.\, X$
2. $\forall X.\, X \to X$
3. $\forall X.\, \forall Y.\, X \to Y \to X$
4. $\forall X.\, X \to \mathtt{nat}$

The informal answer is: a polymorphic program must act uniformly in the unknown type. We first look at functions definable in simply typed λ-calculus.

Each term $\Gamma \vdash t : A$ can be interpreted as a set-theoretic function

$$[\![t]\!] : [\![\Gamma]\!] \to [\![A]\!].$$

Assign a set O_X to each type variable $X \in \mathbb{V}$, then extend it to all types:

$$[\![X]\!] = O_X,$$
$$[\![A \to B]\!] = [\![A]\!] \to [\![B]\!],$$

and to contexts:

$$[\![\cdot]\!] = \{*\},$$
$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!].$$

The interpretation of terms is defined inductively, modulo α-equivalence:

$$\begin{aligned}
[\![\Gamma \vdash x_i : A]\!](\rho) &= \rho(i), \\
[\![\Gamma \vdash t\ u : B]\!](\rho) &= [\![\Gamma \vdash t : A \to B]\!](\rho)\,([\![\Gamma \vdash u : A]\!](\rho))\,, \\
[\![\Gamma \vdash \lambda x.\, t : A \to B]\!](\rho) &= (v \mapsto [\![\Gamma, x : A \vdash t : B]\!](\rho, v))\,.
\end{aligned}$$

Here $\rho \in [\![\Gamma]\!]$ is called an *environment*. For $[\![\cdot \vdash t : A]\!](*)$, we simply write $[\![t]\!]$.

Definition 9

A set-theoretic function $f : [\![A]\!] \to [\![B]\!]$ is λ-definable at type $A \to B$ if there is a closed term $t : A \to B$ such that $f = [\![t]\!]$.

Let there be one base type variable X, interpreted by

$$O_X = \{\mathsf{t}, \mathsf{f}\}.$$

The semantic function space contains four functions $O_X \to O_X$:

1. identity: $a \mapsto a$;
2. constant true: $a \mapsto \mathsf{t}$;
3. constant false: $a \mapsto \mathsf{f}$;
4. negation: $\mathsf{t} \mapsto \mathsf{f}$ and $\mathsf{f} \mapsto \mathsf{t}$.

The question is not whether these set-theoretic functions exist. They do. The question is which of them are denotations of closed λ-terms of type $X \to X$.

The set-theoretic interpretation tells us which functions exist.

But we want to know which functions are *definable* by programs.

Logical relations give a test:

> *A definable function must preserve every relation compatible with its type.*

To show that a set-theoretic function is not definable, find a relation that it fails to preserve.

Idea

If v_1 and v_2 are related, then a definable function must send them to related outputs.

A family $\{R^A \subseteq [\![A]\!] \times [\![A]\!]\}_{A:\mathsf{Type}}$ of binary relations is logical if

$$R^{A\to B}(f_1, f_2) \quad \text{iff} \quad \forall x_1\, x_2.\, R^A(x_1, x_2) \implies R^B(f_1(x_1), f_2(x_2)).$$

N.B. A logical relation is determined by R^X for type variables X.

What is $R^{X \to X}$ if ...

1. $R^X = \varnothing$?
2. $R^X = O_X \times O_X$?
3. $R^X = \{(\mathsf{t}, \mathsf{f})\}$?

Let $O_X = \{\mathsf{t}, \mathsf{f}\}$ and $R^X = \{(\mathsf{f}, \mathsf{t})\}$.

Which of the following functions preserve R^X?

1. identity;
2. constant true;
3. constant false;
4. negation.

Theorem 10 (Fundamental Theorem of Logical Relations)

Let $\{R^A\}_{A:\mathsf{Type}}$ be a logical relation. For every typing derivation $\Gamma \vdash t : A$ and every pair of environments $\rho_1, \rho_2 \in [\![\Gamma]\!]$ that are pointwise related, we have

$$R^A([\![\Gamma \vdash t : A]\!](\rho_1), [\![\Gamma \vdash t : A]\!](\rho_2)).$$

Pointwise related means that $R^{A_i}(\rho_1(i), \rho_2(i))$ for every $x_i : A_i \in \Gamma$.

Proof sketch.

By induction on the typing derivation of $\Gamma \vdash t : A$. □

In particular, $R^A([\![t]\!], [\![t]\!])$ for any closed term t of type A.

Consider $O_X = \{\mathsf{t}, \mathsf{f}\}$ and the logical relation determined by

$$R^X = \{(\mathsf{f}, \mathsf{t})\}.$$

Proof.

Suppose that the constant true function $c_{\mathsf{t}}(x) = \mathsf{t}$ is λ-definable. Then there is a closed term $t : X \to X$ with $[\![t]\!] = c_{\mathsf{t}}$.

By the fundamental theorem,

$$R^{X \to X}([\![t]\!], [\![t]\!]).$$

Since $R^X(\mathsf{f}, \mathsf{t})$, the arrow clause gives

$$R^X([\![t]\!](\mathsf{f}), [\![t]\!](\mathsf{t})).$$

But $[\![t]\!]$ is constant true, so this says $R^X(\mathsf{t}, \mathsf{t})$, a contradiction. □

Exercises

1. Show that the constant function $f(x) = \mathsf{f}$ is not λ-definable.
2. Show that the negation function $\neg$ is not λ-definable.

For simply typed terms, logical relations tell us:

definable programs preserve relations.

For polymorphic terms, the relation for a type variable is not fixed.

$X \quad \rightsquigarrow \quad$ any relation chosen by the user

Parametricity says that a polymorphic program must work uniformly for every such choice.

No set-theoretic model for polymorphic λ-calculus

We would like to apply the same approach to polymorphic λ-calculus, but the impredicative universal type makes the naive set-theoretic account circular:

1. the universal quantification $\forall X.\, A$ is impredicative;
2. $[\![\forall X.\, A]\!]$ should depend on $[\![A[B/X]]\!]$ for every $B : \mathtt{Type}$;
3. this includes the case $B = \forall X.\, A$ itself.

In fact, Reynolds showed that polymorphism is not set-theoretic in classical set theory, due to a cardinality obstruction [Reynolds, 1984].

Thus one has to use other models, constructive set theory [Pitts, 1987], or weaker predicative versions of parametric polymorphism [Leivant, 1991].

Following Girard's *reducibility candidates* [Girard et al., 1989], assume a set $\mathcal{U}$ of relation candidates in some model. A family $\{R^A_\Phi\}_{\Delta \vdash A}$ is logical if

$$
\begin{aligned}
R^X_\Phi(x_1, x_2) \quad &\text{iff} \quad \Phi(X)(x_1, x_2), \\
R^{A \to B}_\Phi(f_1, f_2) \quad &\text{iff} \quad \forall x_1\, x_2.\, R^A_\Phi(x_1, x_2) \implies R^B_\Phi(f_1(x_1), f_2(x_2)), \\
R^{\forall X.\, A}_\Phi(p_1, p_2) \quad &\text{iff} \quad \forall U \in \mathcal{U}.\, R^A_{\Phi; X \mapsto U}(p_1(U), p_2(U)).
\end{aligned}
$$

Here $\Phi : \Delta \to \mathcal{U}$ maps type variables to relation candidates, and $\Phi; X \mapsto U$ maps X to U and maps every other variable Y to $\Phi(Y)$.

If Δ is empty, the subscript Φ in R^A_Φ is omitted.

Theorem 11

The fundamental theorem holds for these logical relations. In particular, $R^A(\llbracket t \rrbracket, \llbracket t \rrbracket)$ holds for every closed polymorphic term $t : A$.

Example: $\forall X.\, X$

The type $\forall X.\, X$ is not inhabited.

Suppose that $\vdash t : \forall X.\, X$. Then, by the fundamental theorem,

$$R^{\forall X.\, X}(\llbracket t \rrbracket, \llbracket t \rrbracket).$$

By the definition of the relation at a universal type, this means

$$\forall U \in \mathcal{U}.\, R^{X}_{X \mapsto U}(\llbracket t \rrbracket(U), \llbracket t \rrbracket(U)),$$

or equivalently,

$$\forall U \in \mathcal{U}.\, U(\llbracket t \rrbracket(U), \llbracket t \rrbracket(U)).$$

Choosing U to be the empty relation gives

$$(\llbracket t \rrbracket(U), \llbracket t \rrbracket(U)) \in \varnothing,$$

a contradiction. Hence there is no closed term of type $\forall X.\, X$.

Exercise

Use parametricity to explain why every closed term of type

$$\forall X.\, X \to X$$

behaves like the identity.

Hint: choose a relation containing exactly one pair (a, b).

Instantiate the relation candidate for X with the graph relation

$$R^X = \{(a, f(a)) \mid a \in [\![A]\!]\}$$

for some function $f : [\![A]\!] \to [\![B]\!]$.

Applying the fundamental theorem yields, for any $t : \forall X.\, \mathtt{list}(X) \to \mathtt{list}(X)$, the commutative diagram

$$\begin{array}{ccc} [\![\mathtt{list}(A)]\!] & \xrightarrow{[\![t]\!]_A} & [\![\mathtt{list}(A)]\!] \\ {\scriptstyle \mathtt{map}\ f}\downarrow & & \downarrow{\scriptstyle \mathtt{map}\ f} \\ [\![\mathtt{list}(B)]\!] & \xrightarrow[[\![t]\!]_B]{} & [\![\mathtt{list}(B)]\!] \end{array}$$

N.B. The equation is derived in the working model, so it does not necessarily imply $=_\beta$ between λ-terms.

This specialised form of the fundamental theorem is known as a *free theorem* [Wadler, 1989].

Summary

1. System F adds type abstraction and type application to simply typed λ-calculus.
2. Universal types support uniform programs such as identity, projections, and Church encodings.
3. Type safety and strong normalisation still hold.
4. Parametricity says that polymorphic programs preserve relations, which gives equations ‘for free’ from types.

This is the point at which type theory begins to act not only as a discipline for programs, but also as a method for reasoning about what programs can be.

References I


Girard, J.-Y., Lafont, Y., and Taylor, P. (1989).
Proofs and Types.
Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.


Leivant, D. (1991).
Finitely stratified polymorphism.
Information and Computation, 93(1):93–113.


Pitts, A. M. (1987).
Polymorphism is set theoretic, constructively.
In Pitt, D. H., Poigné, A., and Rydeheard, D. E., editors, *Category Theory and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 12–39. Springer, Berlin, Heidelberg.


Reynolds, J. C. (1984).
Polymorphism is not set-theoretic.
In Kahn, G., MacQueen, D. B., and Plotkin, G., editors, *Semantics of Data Types (SDT)*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156.

Wadler, P. (1989).
Theorems for free!
In *4th International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, New York, NY, USA. ACM Press.