

λ -CALCULUS

PARAMETRIC POLYMORPHISM

陳亮廷 Chen, Liang-Ting

Formosan Summer School on Logic, Language, and Computation 2024

Institute of Information Science
Academia Sinica

POLYMORPHIC λ -CALCULUS: STATIC

Given a set \mathbb{V} of type variables, the judgement $A : \text{Type}$ is defined by defined by

$$\frac{}{X : \text{Type}} \text{ (tvar), if } X \in \mathbb{V}$$

$$\frac{A : \text{Type} \quad B : \text{Type}}{A \rightarrow B : \text{Type}} \text{ (fun)}$$

$$\frac{A : \text{Type} \quad X \in \mathbb{V}}{\forall X. A : \text{Type}} \text{ (universal)}$$

where X may or may not occur in A .

The polymorphic type $\forall X. A$ provides a universal type for every type B by instantiating X for B , i.e. $A[B/x]$.

For example, the polymorphic type allows us to express terms that should work on arbitrary types, such as

- $\text{id} : \forall X. X \rightarrow X$
- $\text{proj}_1 : \forall X. \forall Y. X \rightarrow Y \rightarrow X$
- $\text{proj}_2 : \forall X. \forall Y. X \rightarrow Y \rightarrow Y$
- $\text{length} : \forall X. \text{list } X \rightarrow \text{nat}$
- $\text{singleton} : \forall X. X \rightarrow \text{list}(X)$

Definition 1

The *free variable* $\mathbf{FV}(A)$ of A is defined inductively by

$$\mathbf{FV}(X) = X$$

$$\mathbf{FV}(A \rightarrow B) = \mathbf{FV}(A) \cup \mathbf{FV}(B)$$

$$\mathbf{FV}(\forall X. A) = \mathbf{FV}(A) - \{X\}$$

The function extends to contexts: $\mathbf{FV}(\Gamma) = \{X \in \mathbb{V} \mid \exists(x : A) \in \Gamma \wedge X \in \mathbf{FV}(A)\}$.

Exercise

1. $\mathbf{FV}(\forall X. (X \rightarrow X) \rightarrow X \rightarrow X)$
2. $\mathbf{FV}(x : X_1, y : X_2, z : \forall X. X)$

Permutation of type variables and α -equivalence between types are defined similarly. In particular, the substitution is defined to avoid any capture of free type variables:

Definition 2

The *capture-avoiding substitution* $_{-}[A/X]$ of a type A for a type variable X is defined by

$$\begin{aligned} X[A/X] &= A \\ Y[A/X] &= Y && \text{if } X \neq Y \\ (B \rightarrow C)[A/X] &= (B[A/X]) \rightarrow (C[A/X]) \\ (\forall Y. B)[A/X] &= \forall Y. B[A/X] && \text{if } Y \neq X, Y \notin \mathbf{FV}(A) \end{aligned}$$

Terms in polymorphic λ -calculus are extended with types. We define the set of terms from scratch here:

Definition 3

The set $\Lambda_{\forall}(V, \mathbb{V})$ of terms in polymorphic λ -calculus is defined inductively:

variable $x \in \Lambda_{\forall}(V, \mathbb{V})$ if x is in V

application $t@u \in \Lambda_{\forall}(V, \mathbb{V})$ if $t, u \in \Lambda_{\forall}(V, \mathbb{V})$

abstraction $\lambda(x : A).t$ if $x \in V$, A is a type, and $t \in \Lambda_{\forall}(V, \mathbb{V})$

type abstraction $\lambda X.t$ is in $\Lambda_{\forall}(V, \mathbb{V})$ if X is in \mathbb{V} and t is in $\Lambda_{\forall}(V, \mathbb{V})$

type application $t A$ is in $\Lambda_{\forall}(V, \mathbb{V})$ if t is in $\Lambda_{\forall}(V, \mathbb{V})$ and A is a type.

N.B. $\lambda(x : A).t$ includes the type of x as part of term. We have additionally a *substitution* $t[A/X]$ of a type A for a type variable X in t .

Polymorphic λ -calculus has two kinds of typing judgements.

- $\Delta \vdash A$ stands for a type A under the type context Δ ;
- $\Delta; \Gamma \vdash t : A$ stands for a term t of type A under the context Γ and the type context Δ

where a *type context* is a sequence of type variable X_1, X_2, \dots, X_n .

The new context Δ is used to keep track of type variables available within the term, as they may be introduced by type abstraction.

The judgement $\Delta \vdash A$ is constructed inductively by following rules.

$$\frac{}{\Delta \vdash X} \text{ if } \Delta \ni X$$

$$\frac{\Delta \vdash X \quad \Delta \vdash Y}{\Delta \vdash X \rightarrow Y}$$

$$\frac{\Delta, X \vdash A}{\Delta \vdash \forall X. A}$$

Exercise

Derive the judgement

$$X \vdash X \rightarrow X$$

The judgement $\Delta; \Gamma \vdash t : A$ is defined inductively by following rules.

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

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

$$\frac{\Delta; \Gamma \vdash t : A \rightarrow B \quad \Delta; \Gamma \vdash u : A}{\Delta; \Gamma \vdash t u : B}$$

$$\frac{\Delta \vdash A \quad \Delta; \Gamma, x : A \vdash t : B}{\Delta; \Gamma \vdash \lambda(x : A). t : A \rightarrow B}$$

$$\frac{\Delta; \Gamma \vdash t : \forall X. A \quad \Delta \vdash B}{\Delta; \Gamma \vdash t B : A[B/x]} \text{ (\forall-elim)}$$

Theorem 4 (Type safety)

Suppose $\Delta; \Gamma \vdash t : A$. Then,

1. $t \rightarrow_{\beta} u$ implies $\Delta; \Gamma \vdash u : A$;
2. t is in normal form or there exists u such that $t \rightarrow_{\beta} u$

Theorem 5 (Wells, 1999)

It is undecidable whether, given a closed term t of the untyped lambda-calculus, there is a well-typed term t' in polymorphic λ -calculus such that $|t'| = t$.

Two ways to retain decidable type inference:

1. Limit the expressiveness so that type inference remains decidable. For example, *Hindley-Milner type system* adapted by Haskell 98, Standard ML, etc. supports only a limited form of polymorphism but type inference is decidable.
2. Adopt *partial* type inference so that type annotations can be used for, e.g. top-level definitions and local definitions.

Check out *bidirectional type synthesis*.

The typing judgement $\vdash \lambda X. \lambda(x : X). x : \forall X. X \rightarrow X$ is derivable

$$\frac{\frac{\frac{\overline{X \vdash X} \quad \overline{X; x : X \vdash x : X}}{X; \cdot \vdash \lambda(x : X). x : X \rightarrow X}}{\vdash \lambda X. \lambda(x : X). x : \forall X. X \rightarrow X}}$$

Convention 6

$\vdash t : A$ stands for $\cdot \vdash t : \tau$ where both contexts are empty.

Derive the following judgements:

$$1. \vdash (\lambda X Y. \lambda(x : X). \lambda(y : Y). x) : \forall X. \forall Y. X \rightarrow Y \rightarrow X$$

$$2. \vdash \lambda X. \lambda(f : X \rightarrow X). \lambda(x : X). f (f x) : \forall X. (X \rightarrow X) \rightarrow X \rightarrow X$$

Hint. polymorphic λ -calculus F is syntax-directed, so the type inversion holds.

POLYMORPHIC λ -CALCULUS: DYNAMICS AND PROGRAMMING

β -reduction for polymorphic λ -calculus has two rules apart from other structural rules:

$$(\lambda(x : A). t) u \longrightarrow_{\beta} t[u/x] \quad \text{and} \quad (\lambda X. t) A \longrightarrow_{\beta} t[A/X]$$

For example,

$$(\lambda X. \lambda(x : X). x) A t \longrightarrow_{\beta} (\lambda(x : X). x)[A/X] t \equiv (\lambda x : A. x) t \longrightarrow_{\beta} t$$

Similarly, β -reduction extends to subterms of a given term, introducing relations \longrightarrow_{β} and $\twoheadrightarrow_{\beta}$ in the same way.

Definition 7

The *empty type* is defined by

$$\perp := \forall X. X$$

No closed term t has this type! (Why?)

Exercise

Suppose that $\vdash t : \forall X. X$. Can we derive a contradiction?

Definition 8

The *sum type* is defined by

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

It has two injection functions: the first injection is defined by

$$\begin{aligned} \text{left}_{A+B} &:= \lambda(x:A). \lambda X. \lambda(f:A \rightarrow X). \lambda(g:B \rightarrow X). f\ x \\ \text{right}_{A+B} &:= \lambda(y:B). \lambda X. \lambda(f:A \rightarrow X). \lambda(g:B \rightarrow X). g\ y \end{aligned}$$

Exercise

Define

$$\text{either} : \forall X. (A \rightarrow X) \rightarrow (B \rightarrow X) \rightarrow A + B \rightarrow X$$

Definition 9 (Product Type)

The product type is defined by

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

The pairing function is defined by

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

Exercise

Define projections

$$\text{proj}_1 : A \times B \rightarrow A \quad \text{and} \quad \text{proj}_2 : A \times B \rightarrow B$$

The type of Church numerals is defined by

$$\text{nat} := \forall X. (X \rightarrow X) \rightarrow X \rightarrow X$$

Church numerals

$$\mathbf{c}_n : \text{nat}$$

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

Successor

$$\text{suc} : \text{nat} \rightarrow \text{nat}$$

$$\text{suc} := \lambda (n : \text{nat}). \lambda X. \lambda (f : X \rightarrow X). \lambda (x : X). f (n X f x)$$

Addition

$$\text{add} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$$
$$\text{add} := \lambda(n : \text{nat}). \lambda(m : \text{nat}). \lambda X. \lambda(f : X \rightarrow X). \lambda(x : X). \\ (m \ X \ f) (n \ X \ f \ x)$$

Multiplication

$$\text{mul} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$$
$$\text{mul} := ?$$

Conditional

$$\text{ifz} : \forall X. \text{nat} \rightarrow X \rightarrow X \rightarrow X$$

$$\text{ifz} := ?$$

Polymorphic λ -calculus allows us to define *recursor* like `fold` in Haskell.

$$\text{fold}_{\text{nat}} : \forall X. (X \rightarrow X) \rightarrow X \rightarrow \text{nat} \rightarrow X$$

$$\text{fold}_{\text{nat}} := \lambda X. \lambda (f : X \rightarrow X). \lambda (e_0 : X). \lambda (n : \text{nat}). n \ X \ f \ e_0$$

Exercise

Define `add` and `mul` using `foldnat` and justify your answer.

1. `add' := ? : nat → nat → nat`

2. `mul' := ? : nat → nat → nat`

Definition 10

For any type A , the type of lists over A is

$$\text{list}(A) := \forall X. X \rightarrow (A \rightarrow X \rightarrow X) \rightarrow X$$

with list constructors:

$$\text{nil}_A := \lambda X. \lambda(h: X). \lambda(f: A \rightarrow X \rightarrow X). h$$

and cons_A of type $A \rightarrow \text{list}(A) \rightarrow \text{list}(A)$ defined as

$$\lambda(x: A). \lambda(xs: \text{list}(A)). \lambda X. \lambda(h: X). \lambda(f: A \rightarrow X \rightarrow X). f x (xs X h f)$$

Inductive types can be defined in polymorphic λ -calculus [Böhm and Berarducci, 1985], including the empty type, the types of sums, natural numbers, and lists.

The Church encoding shows the expressiveness of polymorphic λ -calculus but is not efficient [Koopman et al., 2014]. Other styles of encoding have been proposed [Firsov et al., 2018] to improve the efficiency and the size and used in implementations.

REASONING WITH TYPES

The *type discipline* of a language does not only check if a program makes sense but also enforces safety properties such as *type safety* and *strong normalisation*.

In fact, types can be used to tell what functions are *definable* or what equations a term should satisfy with respect to a given type.

What terms can be defined for the following types?

1. $\forall X. X$
2. $\forall X. X \rightarrow X$
3. $\forall XY. X \rightarrow Y \rightarrow X$
4. $\forall X. X \rightarrow \text{nat}$

Let's start with functions definable in simply typed λ -calculus first.

Idea

Each term $\Gamma \vdash t : A$ can be interpreted as a *set-theoretic* function f to $\llbracket A \rrbracket$, a designated interpretation of A , from $\llbracket \Gamma \rrbracket = \prod_{x:A \in \Gamma} \llbracket A \rrbracket$.

In detail, we assign a set O_X to each $X \in \mathbb{V}$ and then extend the interpretation to all types:

$$\begin{aligned}\llbracket X \rrbracket &= O_X \\ \llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket\end{aligned}$$

as well as contexts Γ :

$$\begin{aligned} \llbracket \cdot \rrbracket &= \{*\} \\ \llbracket \Gamma, x : A \rrbracket &= \llbracket \Gamma \rrbracket \times \llbracket A \rrbracket. \end{aligned}$$

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

$$\llbracket t \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$$

defined inductively (modulo α -equivalence) by

$$\begin{aligned} \llbracket \Gamma \vdash x_i : A \rrbracket(\rho) &= \rho(i) \\ \llbracket \Gamma \vdash t u : B \rrbracket(\rho) &= \llbracket \Gamma \vdash t : A \rightarrow B \rrbracket(\rho) (\llbracket \Gamma \vdash u : A \rrbracket(\rho)) \\ \llbracket \Gamma \vdash \lambda x. t : A \rightarrow B \rrbracket(\rho) &= (v \mapsto \llbracket \Gamma, x : A \vdash t \rrbracket(\rho, v)) \end{aligned}$$

where $\rho \in \llbracket \Gamma \rrbracket$ is called an *environment*.

N.B. For $\llbracket \cdot \vdash t : A \rrbracket (*)$ we simply write $\llbracket t \rrbracket$.

Definition 11

A set-theoretic function $f: X \rightarrow Y$ is **λ -definable** w.r.t. some interpretation if there is a closed term $t: A \rightarrow B$ such that $f = \llbracket t \rrbracket$.

Suppose that there is only one type variable X and $O_X = \{\mathbf{t}, \mathbf{f}\}$.

Which of the following functions $f: O_X \rightarrow O_X$ are λ -definable?

1. the identity function $f(x) = x$
2. the constant function $f(x) = \mathbf{t}$
3. the constant function $f(x) = \mathbf{f}$
4. the negation function $f(\mathbf{t}) = \mathbf{f}$ and $f(\mathbf{f}) = \mathbf{t}$

Idea

If v_1 and v_2 are related, $\llbracket t \rrbracket(v_1)$ and $\llbracket t \rrbracket(v_2)$ should also be related.

A family $\{R^A \subseteq \llbracket A \rrbracket \times \llbracket A \rrbracket\}_{A:\text{Type}}$ of binary relations is **logical** if

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

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

Exercise

What is $R^{X \rightarrow X}$, if ...

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

Theorem 12 (Fundamental Theorem of Logical Relations)

Let $\{R^A\}_{A:\text{Type}}$ be a logical relation. Then,

$$R^A(\llbracket \Gamma \vdash t : A \rrbracket(\rho_1), \llbracket \Gamma \vdash t : A \rrbracket(\rho_2))$$

for every $\Gamma \vdash t : A$ and environments $\rho_1, \rho_2 \in \llbracket \Gamma \rrbracket$ satisfying $R^{A_i}(\rho_1(i), \rho_2(i))$ for every $x_i : A_i \in \Gamma$.

Proof sketch.

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

In particular, $R^A(\llbracket t \rrbracket, \llbracket t \rrbracket)$ for any closed term t of type A .

Consider $O_X = \{t, f\}$ and the logical relation $\{R^A\}_A$ determined by

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

1. Suppose that the constant function $f(x) = t$ is λ -definable, then $R^{X \rightarrow X}(\llbracket t \rrbracket, \llbracket t \rrbracket)$ by the fundamental theorem. By definition of being logical $R^X(\llbracket t \rrbracket(f), \llbracket t \rrbracket(t))$, i.e. $R^X(t, t)$ —a contradiction. That is, $f(x) = t$ is not λ -definable.

Exercise

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

We would like to apply the same approach of arguing λ -definability to polymorphic λ -calculus, but it is apparently circular:

1. the universal quantification $\forall X.A$ is **impredicative** and
2. $\llbracket \forall X.A \rrbracket$ should depend on $\llbracket A[B/X] \rrbracket$ for any $B : \text{Type}$,
3. including $B = \forall X.A$.

In fact, there is no set-theoretic interpretation for polymorphic λ -calculus [Reynolds, 1984] in classical set theory, due to the *cardinality issue*.

Thus, we have to consider *other models* rather than sets, some constructive set theory [Pitts, 1987], or a weaker but predicative version of parametric polymorphism [Leivant, 1991].

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

$$\begin{aligned} R_{\Phi}^X(x_1, x_2) & \text{ iff } \Phi(X)(x_1, x_2) \\ R_{\Phi}^{A \rightarrow B}(f_1, f_2) & \text{ iff } \forall x_1 x_2. R_{\Phi}^A(x_1, x_2) \implies R_{\Phi}^B(f_1(x_1), f_2(x_2)) \\ R_{\Phi}^{\forall X. A}(x_1, x_2) & \text{ iff } \forall U \in \mathcal{U}. R_{\Phi; X \mapsto U}^A(x_1, x_2) \end{aligned}$$

where $\Phi: \Delta \rightarrow \mathcal{U}$ is a map and $\Phi; X \mapsto U$ means a map s.t. Y is mapped to U if $Y = X$ or $\Phi(Y)$ otherwise.

If Δ is empty, then the subscript Φ in R_{Φ}^A is omitted, i.e. R^A instead.

Theorem 13

The fundamental theorem holds for logical relations i.e. $R^A(\llbracket t \rrbracket, \llbracket t \rrbracket)$ holds for any closed term t of type A in polymorphic λ -calculus.

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 definition, $R^{\forall X. X}(\llbracket t \rrbracket, \llbracket t \rrbracket)$ if and only if

$$\forall U \in \mathcal{U}. R_{X \mapsto U}^X(\llbracket t \rrbracket, \llbracket t \rrbracket) \quad \text{or equivalently,} \quad \forall U \in \mathcal{U}. U(\llbracket t \rrbracket, \llbracket t \rrbracket)$$

Choosing U to be the empty relation \emptyset ,

$$(\llbracket t \rrbracket, \llbracket t \rrbracket) \in \emptyset,$$

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

Consider the case that R^X is instantiated as $\{ (x, f(x)) \mid x \in A \}$ of some $f: A \rightarrow B$ and apply the fundamental theorem to derive, e.g.,

- the following equation for any $t: \forall X. \text{list}(X) \rightarrow \text{list}(X)$:





$$\begin{array}{ccc}
 \llbracket \text{list}(A) \rrbracket & \xrightarrow{\llbracket t \rrbracket_A} & \llbracket \text{list}(A) \rrbracket \\
 \text{map } f \downarrow & & \downarrow \text{map } f \\
 \llbracket \text{list}(B) \rrbracket & \xrightarrow{\llbracket t \rrbracket_B} & \llbracket \text{list}(B) \rrbracket
 \end{array}$$





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

The fundamental theorem is well known for this specialised form, dubbed as *free theorems* [Wadler, 1989].

1. (2.5%) Define $\text{length}_\sigma : \text{list } \sigma \rightarrow \text{nat}$ calculating the length of a list in polymorphic λ -calculus.
2. (5%) Prove Theorem 12.

I would like to thank 薛盛安, 曾顯恩, 蔡子濬 (Gene Tsai), 游書泓, and many other students at FLOLAC for their efforts in reading and correcting the material carefully.

-  **Böhm, C. and Berarducci, A. (1985).**
Automatic synthesis of typed Λ -programs on term algebras.
Theoretical Computer Science, 39:135–154.
-  **Firsov, D., Richard, B., and Stump, A. (2018).**
Efficient Mendler-style lambda-encodings in Cedille.
In Avigad, J. and Mahboubi, A., editors, *Interactive Theorem Proving (ITP)*, volume 10895 of *Lecture Notes in Computer Science*, pages 235–252. Springer, Cham.
-  **Girard, J.-Y., Lafont, Y., and Taylor, P. (1989).**
Proofs and Types.
Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.
-  **Koopman, P., Plasmeijer, R., and Jansen, J. M. (2014).**
Church encoding of data types considered harmful for implementations: Functional pearl.
In *Implementation and Application of Functional Languages (IFL)*, New York, NY, USA.
Association for Computing Machinery.

-  **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.