



Design Principles of Programming Languages

编程语言的设计原理

Haiyan Zhao, Di Wang

赵海燕, 王迪

Peking University, Spring Term 2026



Type Inference

类型推导



Type Erasure & Inference for System F

$$\text{erase}(x) \stackrel{\text{def}}{=} x$$

$$\text{erase}(\lambda x:T_1. t_2) \stackrel{\text{def}}{=} \lambda x. \text{erase}(t_2)$$

$$\text{erase}(t_1 t_2) \stackrel{\text{def}}{=} \text{erase}(t_1) \text{erase}(t_2)$$

$$\text{erase}(\lambda X. t_2) \stackrel{\text{def}}{=} \text{erase}(t_2)$$

$$\text{erase}(t_1 [T_2]) \stackrel{\text{def}}{=} \text{erase}(t_1)$$

Definition (Type Inference)

Given an untyped term m , whether we can find some well-typed term t such that $\text{erase}(t) = m$.

Theorem (Wells, 1994¹)

Type inference for System F is **undecidable**.

¹J. B. Wells. 1994. Typability and Type Checking in the Second-Order λ -Calculus Are Equivalent and Undecidable. In *Logic in Computer Science (LICS'94)*, 176–185. DOI: 10.1109/LICS.1994.316068.



Partial Erasure & Inference for System F

$$\text{erase}_p(x) \stackrel{\text{def}}{=} x$$

$$\text{erase}_p(\lambda x:T_1. t_2) \stackrel{\text{def}}{=} \lambda x:T_1. \text{erase}_p(t_2)$$

$$\text{erase}_p(t_1 t_2) \stackrel{\text{def}}{=} \text{erase}_p(t_1) \text{erase}_p(t_2)$$

$$\text{erase}_p(\lambda X. t_2) \stackrel{\text{def}}{=} \lambda X. \text{erase}_p(t_2)$$

$$\text{erase}_p(t_1 [T_2]) \stackrel{\text{def}}{=} \text{erase}_p(t_1) []$$

Theorem (Boehm 1985², 1989³)

It is **undecidable** whether, given a closed term s in which type applications are marked but the arguments are omitted, there is some well-typed System-F term t such that $\text{erase}_p(t) = s$.

²H.-J. Boehm. 1985. Partial Polymorphic Type Inference is Undecidable. In *Symp. on Foundations of Computer Science (SFCS'85)*, 339–345. DOI: 10.1109/SFCS.1985.44.

³H.-J. Boehm. 1989. Type Inference in the Presence of Type Abstraction. In *Prog. Lang. Design and Impl. (PLDI'89)*, 192–206. DOI: 10.1145/73141.74835.



Fragments of System F

Prenex Polymorphism

- Type variables range only over quantifier-free types (**monotypes**).
- Quantified types (**polytypes**) are not allowed to appear on the left-hand sides of arrows.

Rank-2 Polymorphism

A type is said to be of rank 2 if no path from its root to a \forall quantifier passes to the left of 2 or more arrows.

$(\forall X.X \rightarrow X) \rightarrow \text{Nat}$	✓
$\text{Nat} \rightarrow ((\forall X.X \rightarrow X) \rightarrow (\text{Nat} \rightarrow \text{Nat}))$	✓
$((\forall X.X \rightarrow X) \rightarrow \text{Nat}) \rightarrow \text{Nat}$	✗

Remark

Prenex polymorphism is a **predicative** and rank-1 fragment of System F.
Type inference for ranks 2 and lower is **decidable**!

Simply-Typed Lambda-Calculus with Type Variables



Syntax

$$t ::= x \mid \lambda x:T. t \mid t t \mid \dots$$
$$v ::= \lambda x:T. t \mid \dots$$
$$T ::= X \mid T \rightarrow T \mid \dots$$
$$\Gamma ::= \emptyset \mid \Gamma, x : T$$

Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-Var}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2} \text{ T-Abs}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ T-App}$$



Type Substitutions

Definition

A type substitution is a finite mapping from type variables to types.

Example

We define $\sigma \stackrel{\text{def}}{=} [X \mapsto \text{Bool}, Y \mapsto \text{U}]$ for the substitution that maps X to Bool and Y to U .

We write $\text{dom}(\cdot)$ for left-hand sides of pairs in a substitution, e.g., $\text{dom}(\sigma) = \{X, Y\}$.

We write $\text{range}(\cdot)$ for the right-hand sides of pairs in a substitution, e.g., $\text{range}(\sigma) = \{\text{Bool}, \text{U}\}$.

Remark

The pairs of a substitution are applied **simultaneously**.

For example, $[X \mapsto \text{Bool}, Y \mapsto X \rightarrow X]$ maps Y to $X \rightarrow X$, not $\text{Bool} \rightarrow \text{Bool}$.



Type Substitutions

Application of a Substitution to Types

$$\sigma(X) \stackrel{\text{def}}{=} \begin{cases} T & \text{if } (X \mapsto T) \in \sigma \\ X & \text{if } X \text{ is not in the domain of } \sigma \end{cases}$$

$$\sigma(\text{Nat}) \stackrel{\text{def}}{=} \text{Nat}$$

$$\sigma(\text{Bool}) \stackrel{\text{def}}{=} \text{Bool}$$

$$\sigma(T_1 \rightarrow T_2) \stackrel{\text{def}}{=} \sigma(T_1) \rightarrow \sigma(T_2)$$

Composition of Substitutions

$$\sigma \circ \gamma \stackrel{\text{def}}{=} \left[\begin{array}{ll} X \mapsto \sigma(T) & \text{for each } (X \mapsto T) \in \gamma \\ X \mapsto T & \text{for each } (X \mapsto T) \in \sigma \text{ with } X \notin \text{dom}(\gamma) \end{array} \right]$$



Type Substitutions

Application of a Substitution to Contexts

$$\sigma(x_1 : T_1, \dots, x_n : T_n) \stackrel{\text{def}}{=} (x_1 : \sigma(T_1), \dots, x_n : \sigma(T_n))$$

Application of a Substitution to Terms

$$\sigma(x) \stackrel{\text{def}}{=} x$$

$$\sigma(\lambda x:T_1. t_2) \stackrel{\text{def}}{=} \lambda x:\sigma(T_1). \sigma(t_2)$$

$$\sigma(t_1 t_2) \stackrel{\text{def}}{=} \sigma(t_1) \sigma(t_2)$$

Theorem (Preservation of Typing under a Substitution)

If σ is any type substitution and $\Gamma \vdash t : T$, then $\sigma(\Gamma) \vdash \sigma(t) : \sigma(T)$.

Type Inference

Definition (Type Inference in terms of Substitutions)

Let Γ be a context and t be a term. **A solution for** (Γ, t) is a pair (σ, T) such that $\sigma(\Gamma) \vdash \sigma(t) : T$.

Remark (Two Views of $\sigma(\Gamma) \vdash \sigma(t) : T$)

- **Type Inference:** does there exist **some** σ such that $\sigma(\Gamma) \vdash \sigma(t) : T$ for some T ?
- Another view: for **every** σ , do we have $\sigma(\Gamma) \vdash \sigma(t) : T$ for some T ?
 - This corresponds to **parametric polymorphism**, e.g., $\emptyset \vdash \lambda f:X \rightarrow X. \lambda a:X. f (f a) : (X \rightarrow X) \rightarrow X \rightarrow X$.

Example

Let $\Gamma \stackrel{\text{def}}{=} f : X, a : Y$ and $t \stackrel{\text{def}}{=} f a$. Below gives some solutions for (Γ, t) :

σ	T	σ	T
$[X \mapsto Y \rightarrow \text{Nat}]$	Nat	$[X \mapsto Y \rightarrow Z]$	Z
$[X \mapsto Y \rightarrow Z, Z \mapsto \text{Nat}]$	Z	$[X \mapsto Y \rightarrow \text{Nat} \rightarrow \text{Nat}]$	$\text{Nat} \rightarrow \text{Nat}$
$[X \mapsto \text{Nat} \rightarrow \text{Nat}, Y \mapsto \text{Nat}]$	Nat		



Erasure (revisited)

$$\text{erase}(x) \stackrel{\text{def}}{=} x$$

$$\text{erase}(\lambda x:T_1. t_2) \stackrel{\text{def}}{=} \lambda x. \text{erase}(t_2)$$

$$\text{erase}(t_1 t_2) \stackrel{\text{def}}{=} \text{erase}(t_1) \text{erase}(t_2)$$

Definition (Type Inference)

Let Γ be a context and m be an untyped term. A solution for (Γ, m) is a substitution (σ, T) such that $\sigma(\Gamma) \vdash m : T$.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-Var}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x. t_2 : T_1 \rightarrow T_2} \text{ T-Abs}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ T-App}$$

Given the derivation, it is trivial to construct a well-typed term t such that $\text{erase}(t) = m$.



Constraint Typing

Definition

A constraint set C is a set of equations $\{S_i = T_i\}^{1 \dots n}$ where S_i 's and T_i 's are types.

$\Gamma \vdash t : T \mid_{\mathcal{X}} C$: “term t has type T under context Γ whenever constraints C are satisfied”

The set \mathcal{X} is used to track **new** type variables introduced in each subderivation.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_{\emptyset} \{ \}} \text{CT-Var}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid_{\mathcal{X}} C}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2 \mid_{\mathcal{X}} C} \text{CT-Abs}$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid_{\mathcal{X}_1} C_1 \quad \Gamma \vdash t_2 : T_2 \mid_{\mathcal{X}_2} C_2 \quad \mathcal{X}_1 \cap \mathcal{X}_2 = \mathcal{X}_1 \cap FV(T_2) = \mathcal{X}_2 \cap FV(T_1) = \emptyset \quad \mathcal{X} \notin \mathcal{X}_1, \mathcal{X}_2, T_1, T_2, C_1, C_2, \Gamma, t_1, t_2 \quad C' = C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow \mathcal{X}\}}{\Gamma \vdash t_1 t_2 : \mathcal{X} \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{\mathcal{X}\}} C'} \text{CT-App}$$

Question (Exercise 22.3.3)

Construct a constraint typing derivation for $\lambda x : X. \lambda y : Y. \lambda z : Z. (x z) (y z)$.



Solutions for Constraint Typing

Definition

A substitution σ is said to **unify** an equation $S = T$ if $\sigma(S) = \sigma(T)$.
We say that σ unifies a constraint set C if it unifies every equation in C .

Definition

Suppose that $\Gamma \vdash t : S \mid_{\mathcal{X}} C$. **A solution for** (Γ, t, S, C) is a pair (σ, T) such that σ unified C and $\sigma(S) = T$.

Remark

Recall that **a solution for** (Γ, t) is a pair (σ, T) such that $\sigma(\Gamma) \vdash \sigma(t) : T$.
What are the relation between the two definitions of solutions for type inference?



Properties of Constraint Typing

Theorem (Soundness)

Suppose that $\Gamma \vdash t : S \mid_{\mathcal{X}} C$. If (σ, T) is a solution for (Γ, t, S, C) , then it is also a solution for (Γ, t) .

Proof Sketch

By induction on the derivation of constraint typing.

Theorem (Completeness)

Suppose $\Gamma \vdash t : S \mid_{\mathcal{X}} C$. If (σ, T) is a solution for (Γ, t) and $\text{dom}(\sigma) \cap \mathcal{X} = \emptyset$, then there is some solution (σ', T) for (Γ, t, S, C) such that $\sigma' \setminus \mathcal{X} = \sigma$.

Proof Sketch

By induction on the derivation of constraint typing.

Unification

Remark

Hindley (1969)⁴ and Milner (1978)⁵ apply unification to calculate a **“best” solution** to a given constraint set.

Definition

A substitution σ is less specific (or **more general**) than a substitution σ' , written $\sigma \sqsubseteq \sigma'$, if $\sigma' = \gamma \circ \sigma$ for some γ .

A **principal unifier** (or sometimes **most general unifier**) for a constraint set C is a substitution σ that unifies C and such that $\sigma \sqsubseteq \sigma'$ for every substitution σ' unifying C .

Question (Exercise 22.4.3)

Write down principal unifiers (when they exist) for the following sets of constraints:

$$\begin{array}{lll} \{X = \text{Nat}, Y = X \rightarrow X\} & \{\text{Nat} \rightarrow \text{Nat} = X \rightarrow Y\} & \{X \rightarrow Y = Y \rightarrow Z, Z = U \rightarrow W\} \\ \{\text{Nat} = \text{Nat} \rightarrow Y\} & \{Y = \text{Nat} \rightarrow Y\} & \{\} \end{array}$$

⁴R. Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. of the American Math. Society*, 146, 29–60. doi: 10.2307/1995158.

⁵R. Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, 17, 348–375, 3. doi: 10.1016/0022-0000(78)90014-4.

Unification Algorithm



$unify(C)$ = if $C = \emptyset$, then $[\]$
else let $\{S = T\} \cup C' = C$ in
 if $S = T$
 then $unify(C')$
 else if $S = X$ and $X \notin FV(T)$
 then $unify([X \mapsto T]C') \circ [X \mapsto T]$
 else if $T = X$ and $X \notin FV(S)$
 then $unify([X \mapsto S]C') \circ [X \mapsto S]$
 else if $S = S_1 \rightarrow S_2$ and $T = T_1 \rightarrow T_2$
 then $unify(C' \cup \{S_1 = T_1, S_2 = T_2\})$
 else
 fail

What if we omit the occur checks (i.e., $X \notin FV(T)$ and $X \notin FV(S)$)?

Correctness of Unification Algorithm



Theorem

The algorithm *unify* always terminates, failing when given a non-unifiable constraint set as input and otherwise returning a principal unifier.

Proof Sketch

- **Termination:** define the **degree** of C to be the pair (number of distinct type variables, total size of types).
- *unify*(C) **returns a unifier:** prove by induction on the number of recursive calls to *unify*.
 - Fact: if σ unifies $[X \mapsto T]D$, then $\sigma \circ [X \mapsto T]$ unifies $\{X = T\} \cup D$.
- *unify*(C) returns a **principal** unifier: prove by induction on the number of recursive calls.



Principal Types

Definition

A **principal solution** for (Γ, t, S, C) is a solution (σ, T) such that, $\sigma \sqsubseteq \sigma'$ for any other solution (σ', T') .
When (σ, T) is a principal solution, we call T a **principal type** of t under Γ .

Theorem

If (Γ, t, S, C) has any solution, then it has a principal one.
The *unify* algorithm can be used to determine if there exists a solution and, if so, to calculate a principal one.

Corollary

It is decidable whether (Γ, t) has a solution.

Remark

Recall that type inference for System F is **undecidable**.



Recall: Prenex Polymorphism

Prenex Polymorphism

- Type variables range only over quantifier-free types (**monotypes**).
- Quantified types (**polytypes**) are not allowed to appear on the left-hand sides of arrows.

Let-Polymorphism is a Variant of Prenex Polymorphism where ...

- Quantifiers can only occur at the outermost level of types.
- Type abstractions are implicitly introduced at **let-bindings**.
- Type applications are implicitly introduced at **variables**.



Let-Polymorphism as a Fragment of System F

Syntax

$t ::= x \mid \lambda x:T. t \mid t t \mid \text{let } x = t \text{ in } t \mid \dots$

$v ::= \lambda x:T. t \mid \dots$

$T ::= X \mid T \rightarrow T \mid \dots$

$\mathbb{T} ::= \forall X_1 \dots X_n. T$

$\Gamma ::= \emptyset \mid \Gamma, x : \mathbb{T}$

Typing

$$\frac{\Gamma \vdash t_1 : T_1 \quad \{X_1, \dots, X_n\} = FV(T_1) \setminus FV(\Gamma) \quad \mathbb{T}_1 = \forall X_1 \dots X_n. T_1 \quad \Gamma, x : \mathbb{T}_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \text{T-Let}$$

$$\frac{x : \forall X_1 \dots X_n. T \in \Gamma}{\Gamma \vdash x : [X_1 \mapsto S_1, \dots, X_n \mapsto S_n]T} \text{T-Var}$$



Let-Polymorphism as a Fragment of System F

Example

let double = $\lambda f:(X \rightarrow X). \lambda a:X. f (f a)$ **in**
{double ($\lambda x:\text{Nat}. \text{succ} (\text{succ } x)$) 1,
double ($\lambda x:\text{Bool}. x$) false}

(T-Let): $\forall X. (X \rightarrow X) \rightarrow X \rightarrow X$

(T-Var): $(\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat}$

(T-Var): $(\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow \text{Bool}$

Observation

Let-polymorphism can be equivalently implemented in simply-typed lambda-calculus with the following rule:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \text{T-LetPoly}$$



Constraint Typing for Let-Polymorphism

$$\frac{\Gamma \vdash t_1 : T_1 \mid x_1 C_1 \quad \{X_1, \dots, X_n\} = FV(T_1) \cup FV(C_1) \setminus FV(\Gamma) \quad \mathbb{T}_1 = \forall X_1 \dots X_n. C_1 \supset T_1 \quad \Gamma, x : \mathbb{T}_1 \vdash t_2 : T_2 \mid x_2 C_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2 \mid x_1 \cup x_2 C_1 \cup C_2} \text{CT-Let}$$

$$\frac{x : \forall X_1 \dots X_n. C \supset T \in \Gamma \quad Y_1, \dots, Y_n \notin X_1, \dots, X_n, T, \Gamma}{\Gamma \vdash x : [X_1 \mapsto Y_1, \dots, X_n \mapsto Y_n]T \mid_{\{Y_1, \dots, Y_n\}} [X_1 \mapsto Y_1, \dots, X_n \mapsto Y_n]C} \text{CT-Var}$$

Example

let double = $\lambda f:(X \rightarrow X). \lambda a:X. f (f a)$ **in**

[CT-Let]: $\forall X, X_1, X_2. \{X \rightarrow X = X \rightarrow X_1, X \rightarrow X = X_1 \rightarrow X_2\} \supset (X \rightarrow X) \rightarrow X \rightarrow X_2 \mid \{\dots\}$

{double ($\lambda x:\text{Nat}. \text{succ} (\text{succ } x)$) 1,

[CT-Var]: $(Y \rightarrow Y) \rightarrow Y \rightarrow Y_2 \mid \{Y \rightarrow Y = Y \rightarrow Y_1, Y \rightarrow Y = Y_1 \rightarrow Y_2\} \cup \{Y \rightarrow Y = \text{Nat} \rightarrow \text{Nat}\}$

double ($\lambda x:\text{Bool}. x$) false}

[CT-Var]: $(Z \rightarrow Z) \rightarrow Z \rightarrow Z_2 \mid \{Z \rightarrow Z = Z \rightarrow Z_1, Z \rightarrow Z = Z_1 \rightarrow Z_2\} \cup \{Z \rightarrow Z = \text{Bool} \rightarrow \text{Bool}\}$



Interaction with Side Effects

Example

Let-polymorphism would assign $\forall X. \text{Ref}(X \rightarrow X)$ to r in the following code:

```
let r = ref ( $\lambda x:X. x$ ) in  
  ( $r := (\lambda x:\text{Nat}. \text{succ } x)$ ;  
   (!r)true);
```

When type-checking the second line, we instantiate r to have type $\text{Ref}(\text{Nat} \rightarrow \text{Nat})$.

When type-checking the third line, we instantiate r to have type $\text{Ref}(\text{Bool} \rightarrow \text{Bool})$.

But this is **unsound!**

Value Restriction

A let-binding can be treated polymorphically—i.e., its free type variables can be generalized—only if its right-hand side is a **syntactic value**.

Homework



Question

Consider the following lambda-abstraction:

$$\lambda x:X. x x$$

Construct a constraint typing derivation for it.

Is the constraint set unifiable?

What if removing the occur checks in the *unify* algorithm and allowing recursive types, as shown below?

What is the result of this *unify* algorithm?

$$\begin{aligned} \text{unify}(C) &= \dots \\ &\text{else if } S = X \text{ and } X \notin FV(T) \\ &\quad \text{then } \text{unify}([X \mapsto T]C') \circ [X \mapsto T] \\ &\text{else if } S = X \text{ and } X \in FV(T) \\ &\quad \text{then } \text{unify}([X \mapsto \mu X. T]C') \circ [X \mapsto \mu X. T] \\ &\dots \end{aligned}$$