编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao, Di Wang
赵海燕，王迪

Peking University, Spring Term 2023

Chap 22: Type Reconstruction

Formulation

Constraint-Based Typing

Unification & Principal Types

Extension with Let-Polymorphism

Recall: Erasure & Type Reconstruction

$$
\begin{aligned}
erase(x) &\stackrel{\text{def}}{=} x \\
erase(\lambda x{:}T_1.\, t_2) &\stackrel{\text{def}}{=} \lambda x.\, erase(t_2) \\
erase(t_1\ t_2) &\stackrel{\text{def}}{=} erase(t_1)\ erase(t_2) \\
erase(\lambda X.\, t_2) &\stackrel{\text{def}}{=} erase(t_2) \\
erase(t_1\ [T_2]) &\stackrel{\text{def}}{=} erase(t_1)
\end{aligned}
$$

Definition (Type Reconstruction)

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

Recall: Prenex Polymorphism

Prenex Polymorphism

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

Remark

Type reconstruction for prenex polymorphism is **decidable**!

In This Chapter

- We first develop a type-reconstruction algorithm for **simply-typed lambda-calculus**.
- We then consider a variant of prenex polymorphism named **let-polymorphism**.

Formulation

Simply-Typed Lambda-Calculus with Type Variables

Syntax

$$
\begin{aligned}
t &::= x \mid \lambda x{:}T.\, t \mid t\ t \mid \ldots \\
v &::= \lambda x{:}T.\, t \mid \ldots \\
T &::= X \mid T \to T \mid \ldots \\
\Gamma &::= \varnothing \mid \Gamma, x : T
\end{aligned}
$$

Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \qquad \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.\, t_2 : T_1 \to T_2}\ \textsc{T-Abs} \qquad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \textsc{T-App}$$

Type Substitutions

Definition

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

Example

We define $\sigma \stackrel{\text{def}}{=} [X \mapsto \texttt{Bool}, Y \mapsto U]$ for the substitution that maps X to $\texttt{Bool}$ and Y to U.
We write $dom(\cdot)$ for left-hand sides of pairs in a substitution, e.g., $dom(\sigma) = \{X, Y\}$.
We write $range(\cdot)$ for the right-hand sides of pairs in a substitution, e.g., $range(\sigma) = \{\texttt{Bool}, U\}$.

Remark

The pairs of a substitution are applied **simultaneously**.
For example, $[X \mapsto \texttt{Bool}, Y \mapsto X \to X]$ maps Y to $X \to X$, not $\texttt{Bool} \to \texttt{Bool}$.

Type Substitutions

Application of a Substitution to Types

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

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

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

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

Composition of Substitutions

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

Type Substitutions

Application of a Substitution to Contexts

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

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 Reconstruction

Definition (Type Reconstruction 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 reconstruction**: 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., $\varnothing \vdash \lambda f{:}X \to X.\ \lambda a{:}X.\ f\ (f\ a) : (X \to X) \to X \to X$.

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
$[X \mapsto Y \to \text{Nat}]$	Nat	$[X \mapsto Y \to Z]$	Z
$[x \mapsto Y \to Z, Z \mapsto \text{Nat}]$	Z	$[X \mapsto Y \to \text{Nat} \to \text{Nat}]$	$\text{Nat} \to \text{Nat}$
$[X \mapsto \text{Nat} \to \text{Nat}, Y \mapsto \text{Nat}]$	Nat		

Erasure (revisited)

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

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

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

Definition (Type Reconstruction)

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{T-Var} \qquad \frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x.\, t_2 : T_1 \to T_2}\ \textsc{T-Abs} \qquad \frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \textsc{T-App}$$

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

Constraint-Based Typing

Constraint Typing

Definition

A constraint set C is a set of equations $\{S_i = T_i{}^{1\ldots n}\}$ 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"

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_{\varnothing} \{\}} \text{ CT-VAR} \qquad \frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid_{\mathcal{X}} C}{\Gamma \vdash \lambda x{:}T_1 .\, t_2 : T_1 \to T_2 \mid_{\mathcal{X}} C} \text{ CT-ABS}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid_{\mathcal{X}_1} C_1 \qquad \Gamma \vdash t_2 : T_2 \mid_{\mathcal{X}_2} C_2 \qquad \mathcal{X}_1 \cap \mathcal{X}_2 = \mathcal{X}_1 \cap FV(T_2) = \mathcal{X}_2 \cap FV(T_1) = \varnothing \\ X \notin \mathcal{X}_1, \mathcal{X}_2, T_1, T_2, C_1, C_2, \Gamma, t_1, t_2 \qquad C' = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}\end{array}}{\Gamma \vdash t_1\, t_2 : X \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\}} C'} \text{ CT-APP}$$

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

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 $\Gamma \vdash t : S \mid C$. **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 reconstruction?

Properties of Constraint Typing

Theorem (Soundness)

Suppose that $\Gamma \vdash t : S \mid C$. 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 $dom(\sigma) \cap \mathcal{X} = \emptyset$, 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 & Principal Types

Find a Most General Substitution σ that Unifies a Constraint Set C

Unification

Remark

Hindley (1969)[1] and Milner (1978)[2] 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:

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

[1] R. 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.

[2] R. 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

$$
\begin{array}{rcl}
\mathit{unify}(C) & = & \text{if } C = \emptyset\text{, then } [\,] \\
& & \text{else let } \{S = T\} \cup C' = C \text{ in} \\
& & \quad \text{if } S = T \\
& & \qquad \text{then } \mathit{unify}(C') \\
& & \quad \text{else if } S = X \text{ and } X \notin FV(T) \\
& & \qquad \text{then } \mathit{unify}([X \mapsto T]C') \circ [X \mapsto T] \\
& & \quad \text{else if } T = X \text{ and } X \notin FV(S) \\
& & \qquad \text{then } \mathit{unify}([X \mapsto S]C') \circ [X \mapsto S] \\
& & \quad \text{else if } S = S_1 \to S_2 \text{ and } T = T_1 \to T_2 \\
& & \qquad \text{then } \mathit{unify}(C' \cup \{S_1 = T_1, S_2 = T_2\}) \\
& & \quad \text{else} \\
& & \qquad \mathit{fail}
\end{array}
$$

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 $[\mathrm{X} \mapsto \mathrm{T}]D$, then $\sigma \circ [\mathrm{X} \mapsto \mathrm{T}]$ unifies $\{\mathrm{X} = \mathrm{T}\} \cup D$.
- $unify(C)$ 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 reconstruction for System F is **undecidable**.

Extension with Let-Polymorphism

Recall: Prenex Polymorphism

Prenex Polymorphism

- Type variables range only over quantifier-free types (**monotypes**).
- Quantified types (**polytypes**) are not allows 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

$$
\begin{aligned}
t &::= x \mid \lambda x{:}T.\,t \mid t\,t \mid \texttt{let}\,x = t\,\texttt{in}\,t \mid \ldots \\
v &::= \lambda x{:}T.\,t \mid \ldots \\
T &::= X \mid T \to T \mid \ldots \\
\mathbb{T} &::= \forall X_1 \ldots X_n.T \\
\Gamma &::= \varnothing \mid \Gamma, x : \mathbb{T}
\end{aligned}
$$

Typing

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

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

Let-Polymorphism as a Fragment of System F

Example

```
let double = λf:(X→X). λa:X. f (f a) in
  {double (λx:Nat. succ (succ x)) 1,
   double (λx:Bool. x) false}
```

(T-LET): $\forall X.(X \to X) \to X \to X$

(T-VAR): $(\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}$

(T-VAR): $(\mathsf{Bool} \to \mathsf{Bool}) \to \mathsf{Bool} \to \mathsf{Bool}$

Observation

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

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2 : T_2}\ \text{T-LETPOLY}$$

Constraint Typing for Let-Polymorphism

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid_{\mathcal{X}_1} C_1 \qquad \{X_1, \ldots, X_n\} = FV(T_1) \cup FV(C_1) \setminus FV(\Gamma) \\ \mathbb{T}_1 = \forall X_1 \ldots X_n . C_1 \supset T_1 \qquad \Gamma, x : \mathbb{T}_1 \vdash t_2 : T_2 \mid_{\mathcal{X}_2} C_2\end{array}}{\Gamma \vdash \mathtt{let}\, x = t_1 \,\mathtt{in}\, t_2 : T_2 \mid_{\mathcal{X}_1 \cup \mathcal{X}_2} C_1 \cup C_2} \text{ CT-LET}$$

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

Example

```
let double = λf:(X→X). λa:X. f (f a) in
```
(CT-LET): $\forall X, X_1, X_2 . \{X \to X = X \to X_1, X \to X = X_1 \to X_2\} \supset (X \to X) \to X \to X_2 \mid \{\ldots\}$
```
  {double (λx:Nat. succ (succ x)) 1,
```
(CT-VAR): $(Y \to Y) \to Y \to Y_2 \mid \{Y \to Y = Y \to Y_1, Y \to Y = Y_1 \to Y_2\} \cup \{Y \to Y = \mathrm{Nat} \to \mathrm{Nat}\}$
```
   double (λx:Bool. x) false}
```
(CT-VAR): $(Z \to Z) \to Z \to Z_2 \mid \{Z \to Z = Z \to Z_1, Z \to Z = Z_1 \to Z_2\} \cup \{Z \to Z = \mathrm{Bool} \to \mathrm{Bool}\}$

Interaction with Side Effects

Example

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

```
let r = ref (λx:X. x) in
(r := (λx:Nat. succ x);
 (!r)true);
```

When type-checking the second line, we instantiate `r` to have type $\texttt{Ref}(\texttt{Nat} \to \texttt{Nat})$.
When type-checking the third line, we instantiate `r` to have type $\texttt{Ref}(\texttt{Bool} \to \texttt{Bool})$.
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:

$$\lambda\ \mathtt{x}:\mathtt{X}.\ \ \mathtt{x}\ \ \mathtt{x}$$

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?

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