# Chapter 22: Type Reconstruction (Type Inference)

Calculating a Principal Type for a Term

Constraint-based Typing

Unification and Principle Types

Extension with let-polymorphism

## Type Variables and Type Substitution

- Type variable

  X → X

- Type substitution: finite mapping from type variables to types.

  σ = [X ➔ Bool, Y ➔ U]

  dom(σ) = {X, Y}
  range(σ) = {Bool, U}

Note: the same variables can be in both the domain and the range.
[X ➔ Bool, Y ➔ X→X]

- Application of type substitution to a type:

$$
\begin{aligned}
\sigma(\mathsf{X}) &= \begin{cases} \mathsf{T} & \text{if } (\mathsf{X} \mapsto \mathsf{T}) \in \sigma \\ \mathsf{X} & \text{if X is not in the domain of } \sigma \end{cases} \\
\sigma(\mathtt{Nat}) &= \mathtt{Nat} \\
\sigma(\mathtt{Bool}) &= \mathtt{Bool} \\
\sigma(\mathsf{T}_1 \to \mathsf{T}_2) &= \sigma\mathsf{T}_1 \to \sigma\mathsf{T}_2
\end{aligned}
$$

- Type substitution composition

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

- Type substitution on contexts:
  - σ(x1:T1,...,xn:Tn) = (x1:σT1,... ,xn:σTn).

- Substitution on Terms:
  - A substitution is applied to a term t by applying it to all types appearing in annotations in t.

- Theorem [Preservation of typing under type substitution]: If σ is any type substitution and Γ ⊢ t : T, then σΓ ⊢ σt : σT.

## Two Views of Type Variables

- View 1: "Are all substitution instances of t well typed?" That is, for every σ, do we have

  $\sigma\Gamma \vdash \sigma t : T$

  for some T?
  - E.g., $\lambda f{:}X\to X.\ \lambda a{:}X.\ f\ (f\ a)$

Parametric polymorphism

- View 2. "Is some substitution instance of t well typed?" That is, can we find a σ such that

  $\sigma\Gamma \vdash \sigma t : T$

  for some T?
  - E.g., $\lambda f{:}Y.\ \lambda a{:}X.\ f\ (f\ a)$

Type reconstruction

## Type Reconstruction

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

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1\to T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11}\to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-APP)}$$

EXAMPLE: Let $\Gamma = \mathtt{f}:\mathtt{X}, \mathtt{a}:\mathtt{Y}$ and $\mathtt{t} = \mathtt{f\ a}$. Then

$([\mathtt{X} \mapsto \mathtt{Y}\to\mathtt{Nat}],\ \mathtt{Nat})$ $\quad$ $([\mathtt{X} \mapsto \mathtt{Y}\to\mathtt{Z}],\ \mathtt{Z})$

$([\mathtt{X} \mapsto \mathtt{Y}\to\mathtt{Z},\ \mathtt{Z} \mapsto \mathtt{Nat}],\ \mathtt{Z})$ $\quad$ $([\mathtt{X} \mapsto \mathtt{Y}\to\mathtt{Nat}\to\mathtt{Nat}],\ \mathtt{Nat}\to\mathtt{Nat})$

$([\mathtt{X} \mapsto \mathtt{Nat}\to\mathtt{Nat},\ \mathtt{Y} \mapsto \mathtt{Nat}],\ \quad \mathtt{Nat})$

are all solutions for $(\Gamma, \mathtt{t})$.

## Constraint-based Typing

The constraint typing relation

$\Gamma \vdash t : T \mid_X C$

is defined as follows.

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T} \mid_{\varnothing} \{\}} \qquad \text{(CT-VAR)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2 \mid_{\mathcal{X}} C}{\Gamma \vdash \lambda \mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1\to\mathtt{T}_2 \mid_{\mathcal{X}} C} \qquad \text{(CT-ABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \mid_{\mathcal{X}_1} C_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 \mid_{\mathcal{X}_2} C_2 \\ \mathcal{X}_1 \cap \mathcal{X}_2 = \mathcal{X}_1 \cap FV(\mathtt{T}_2) = \mathcal{X}_2 \cap FV(\mathtt{T}_1) = \varnothing \\ \mathtt{X} \notin \mathcal{X}_1, \mathcal{X}_2, \mathtt{T}_1, \mathtt{T}_2, C_1, C_2, \Gamma, \mathtt{t}_1, \text{or } \mathtt{t}_2 \\ C' = C_1 \cup C_2 \cup \{\mathtt{T}_1 = \mathtt{T}_2\to\mathtt{X}\}\end{array}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{X} \mid_{\mathcal{X}_1\cup\mathcal{X}_2\cup\{\mathtt{X}\}} C'} \qquad \text{(CT-APP)}$$

Exercise: Construct C from the term λx:X, λy:Y, λz:Z. x z (y z)

- Extended with Boolean Expression

$$\Gamma \vdash \texttt{true} : \texttt{Bool} \ |_{\varnothing} \ \{\} \qquad \text{(CT-TRUE)}$$

$$\Gamma \vdash \texttt{false} : \texttt{Bool} \ |_{\varnothing} \ \{\} \qquad \text{(CT-FALSE)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \ |_{\mathcal{X}_1} \ C_1 \\ \Gamma \vdash \texttt{t}_2 : \texttt{T}_2 \ |_{\mathcal{X}_2} \ C_2 \qquad \Gamma \vdash \texttt{t}_3 : \texttt{T}_3 \ |_{\mathcal{X}_3} \ C_3 \\ \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \text{ nonoverlapping} \\ C' = C_1 \cup C_2 \cup C_3 \cup \{\texttt{T}_1 = \texttt{Bool}, \texttt{T}_2 = \texttt{T}_3\}\end{array}}{\Gamma \vdash \texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}_2 \ |_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3} \ C'} \qquad \text{(CT-IF)}$$

Definition: Suppose that $\Gamma \vdash t : S \mid C$. A solution for $(\Gamma,t,S,C)$ is a pair $(\sigma,T)$ such that $\sigma$ satisfies C and $\sigma S = T$.

Recall:

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

What are the relation between these two solutions?

Theorem [Soundness of constraint typing]: Suppose that $\Gamma \vdash t : T \mid C$. If $(\sigma,\tau)$ is a solution for $(\Gamma,t,T,C)$, then it is also a solution for $(\Gamma,t)$.

Proof. By induction on constraint typing derivation.

Theorem [Completeness of constraint typing]:
Suppose $\Gamma \vdash t : S \mid_X C$.
If $(\sigma,T)$ is a solution for $(\Gamma ,t)$ and $dom(\sigma) \cap X = \emptyset$, then there is some solution $(\sigma',T)$ for $(\Gamma ,t,S,C)$ such that $\sigma' \backslash X = \sigma$.

Proof: By induction on the given constraint typing derivation.

## Unification

- Idea from Hindley (1969) and Milner (1978) for calculating "best" solution to constraint sets.

Definition: A substitution $\sigma$ is less specific (or more general) than a substitution $\sigma'$, written $\sigma \sqsubseteq \sigma'$, if

$$\sigma' = \gamma \circ \sigma$$

for some substitution $\gamma$.

Definition: A principal unifier (or sometimes most general unifier) for a constraint set $C$ is a substitution $\sigma$ that satisfies $C$ and such that $\sigma \sqsubseteq \sigma'$ for every substitution $\sigma'$ satisfying $C$.

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

- $\{X = Nat, Y = X \to X\}$
- $\{Nat \to Nat = X \to Y\}$
- $\{X \to Y = Y \to Z, Z = U \to W\}$
- $\{Nat = Nat \to Y\}$
- $\{Y = Nat \to Y\}$
- $\{\}$

## Unification Algorithm

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

No cyclic

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

Proof.

Termination: define degree of C = (number of distinct type variables, total size of types).

Unify(C) returns a unifier: induction on the number of recursive calls of unify. (Fact: $\sigma$ unifies [X -> T]D, then $\sigma \circ$ [X->T] unifies {X = T}∪D)

It returns a principle unifier: induction on the number of recursive calls.

## Principle Types

- If there is some way to instantiate the type variables in a term, e.g.,

  λx:X. λy:Y. λz:Z. (x z) (y z)

  so that it becomes typable, then there is a most general or principal way of doing so.

Unification Algorithm

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

## Implicit Type Annotation

Type reconstruction allows programmers to completely omit type annotations on lambda-abstractions.

$$\frac{X \notin \mathcal{X} \qquad \Gamma, x{:}X \vdash t_1 : T \;|_{\mathcal{X}}\; C}{\Gamma \vdash \lambda x.t_1 : X \to T \;|_{\mathcal{X} \cup \{X\}}\; C} \quad \text{(CT-AbsInf)}$$

## Let-Polymorphism

- Code Duplication:

```
let doubleNat = λf:Nat→Nat. λa:Nat. f(f(a)) in
let doubleBool = λf:Bool→Bool. λa:Bool. f(f(a)) in
let a = doubleNat (λx:Nat. succ (succ x)) 1 in
let b = doubleBool (λx:Bool. x) false in ...Even
```

- One Attempt

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

This is not typable, since double can only be instantiated once.

- Solution: Unfolding "let" (perform a step of evaluation of let)

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

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \ \ |_{\mathcal{X}}\ C}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2 \ \ |_{\mathcal{X}}\ C} \quad \text{(CT-LetPoly)}$$

let double = λf. λa. f(f(a)) in
let a = double (λx:Nat. succ (succ x)) 1 in
let b = double (λx:Bool. x) false in ...

Typable!

- Issue 1: what happens when the let-bound variable does not appear in the body:

let x = <utter garbage> in 5

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

- Issue 2: Avoid re-typechecking when a let-variable appear many times in let x=t1 in t2.

  1. Find a principle type T1 of t1.
  2. Generalize T1 to a schema ∀X1…Xn.T1.
  3. Extend the context with (x, ∀X1…Xn.T1).
  4. Each time we encounter an occurrence of x in t2, look up its type scheme ∀X1…Xn.T1, generate fresh type variables Y1…Yn to instantiate the type scheme, yielding [X1 -> Y1,.. . , Xn -> Yn]T1, which we use as the type of x

## Homework

22.5.5 EXERCISE [RECOMMENDED, ★★★ ↛]: Combine the constraint generation and unification algorithms from Exercises 22.3.10 and 22.4.6 to build a typechecker that calculates principal types, taking the `reconbase` checker as a starting point. A typical interaction with your typechecker might look like:

```
λx:X. x;
▸ <fun> : X → X
λz:ZZ. λy:YY. z (y true);
▸ <fun> : (?X₀→?X₁) → (Bool→?X₀) → ?X₁
λw:W. if true then false else w false;
▸ <fun> : (Bool→Bool) → Bool
```

Type variables with names like $?X_0$ are automatically generated. □