

## 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

$$X \rightarrow X$$

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

$$\sigma = [X \rightarrow \text{Bool}, Y \rightarrow U]$$

$$\text{dom}(\sigma) = \{X, Y\}$$

$$\text{range}(\sigma) = \{\text{Bool}, U\}$$

Note: the same variables can be in both the domain and the range.

$$[X \rightarrow \text{Bool}, Y \rightarrow X \rightarrow X]$$



- Application of type substitution to a type:

$$\begin{aligned} \sigma(X) &= \begin{cases} T & \text{if } (X \mapsto T) \in \sigma \\ X & \text{if } X \text{ is not in the domain of } \sigma \end{cases} \\ \sigma(\text{Nat}) &= \text{Nat} \\ \sigma(\text{Bool}) &= \text{Bool} \\ \sigma(T_1 \rightarrow T_2) &= \sigma T_1 \rightarrow \sigma T_2 \end{aligned}$$

- Type substitution composition

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



- Type substitution on contexts:
  - $\sigma(x_1:T_1, \dots, x_n:T_n) = (x_1:\sigma T_1, \dots, x_n:\sigma T_n)$ .
- 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  $\sigma$  is any type substitution and  $\Gamma \vdash t : T$ , then  $\sigma \Gamma \vdash \sigma t : \sigma T$ .



## Two Views of Type Variables

- **View 1:** “Are all substitution instances of  $t$  well typed?” That is, for **every**  $\sigma$ , do we have

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

for some  $T$ ?

- E.g.,  $\lambda f:T \rightarrow T. \lambda a:T. f (f a)$

- **View 2.** “Is some substitution instance of  $t$  well typed?” That is, can we **find a**  $\sigma$  such that

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

for some  $T$ ?

- E.g.,  $\lambda f:Y. \lambda a:X. f (f a)$



# Type Reconstruction

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

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T}$$

(T-VAR)

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

(T-ABS)

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}$$

(T-APP)



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

$([X \mapsto Y \rightarrow \text{Nat}], \text{Nat})$                        $([X \mapsto Y \rightarrow Z], Z)$   
 $([X \mapsto Y \rightarrow Z, Z \mapsto \text{Nat}], Z)$              $([X \mapsto Y \rightarrow \text{Nat} \rightarrow \text{Nat}], \text{Nat} \rightarrow \text{Nat})$   
 $([X \mapsto \text{Nat} \rightarrow \text{Nat}, Y \mapsto \text{Nat}], \text{Nat} \rightarrow \text{Nat})$

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



# Constraint-based Typing

The constraint typing relation

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

is defined as follows.

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T \mid_{\emptyset} \{}} \quad \text{(CT-VAR)}$$
$$\frac{\Gamma, x:T_1 \vdash t_2 : T_2 \mid_X C}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2 \mid_X C} \quad \text{(CT-ABS)}$$
$$\frac{\Gamma \vdash t_1 : T_1 \mid_{X_1} C_1 \quad \Gamma \vdash t_2 : T_2 \mid_{X_2} C_2 \quad X_1 \cap X_2 = X_1 \cap FV(T_2) = X_2 \cap FV(T_1) = \emptyset \quad X \notin X_1, X_2, T_1, T_2, C_1, C_2, \Gamma, t_1, \text{ or } t_2 \quad C' = C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow X\}}{\Gamma \vdash t_1 t_2 : X \mid_{X_1 \cup X_2 \cup \{X\}} C'} \quad \text{(CT-APP)}$$





- Extended with Boolean Expression

$$\Gamma \vdash \text{true} : \text{Bool} \quad | \emptyset \quad \{ \} \quad (\text{CT-TRUE})$$

$$\Gamma \vdash \text{false} : \text{Bool} \quad | \emptyset \quad \{ \} \quad (\text{CT-FALSE})$$

$$\Gamma \vdash t_1 : T_1 \quad | x_1 \quad C_1$$

$$\Gamma \vdash t_2 : T_2 \quad | x_2 \quad C_2 \quad \Gamma \vdash t_3 : T_3 \quad | x_3 \quad C_3$$

$$x_1, x_2, x_3 \text{ nonoverlapping}$$

$$C' = C_1 \cup C_2 \cup C_3 \cup \{T_1 = \text{Bool}, T_2 = T_3\}$$


---


$$\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \quad | x_1 \cup x_2 \cup x_3 \quad C' \quad (\text{CT-IF})$$


**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.

- Case CT-Var.

$$\frac{\dots \quad x:T \in \Gamma}{\Gamma \vdash x : T \mid \emptyset \ \{ \}} \quad (\text{CT-VAR})$$

$(\sigma, \tau)$  is a solution  $\Rightarrow \sigma T = \tau \Rightarrow \sigma \Gamma \vdash x : \tau$



- Case CT-Abs.

$$\frac{\dots \quad \Gamma, x:T_1 \vdash t_2 : T_2 \quad |x \ C}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2 \quad |x \ C} \quad (\text{CT-ABS})$$

$(\sigma, \tau)$  is a solution to the above denominator  $\rightarrow \sigma$  meets  $C$   
 By induction,  $(\sigma, \sigma T_2)$  is a solution to the above enumerator  
 That is,  $(\sigma, \sigma T_2)$  is a solution to  $\Gamma, x:T_1 \vdash t_2 : T_2$   
 By T-ABS,  $(\sigma, T_1 \rightarrow T_2)$  is a solution to  $\Gamma \vdash \lambda x:T_1. t_2 : T_2$



- Case CT-App

...

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

$(\sigma, \tau)$  is a solution  $\rightarrow \dots$



**Theorem [Completeness of constraint typing]:**

Suppose  $\Gamma \vdash t : S \mid_X C$ .

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

**Proof:** By induction on the given constraint typing  
derivation.

(Think and read the textbook)



# 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:

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





# Unification Algorithm

```
unify(C) = if C = ∅, then [ ]  
            else let {S = T} ∪ C' = C in  
                if S = T  
                    then unify(C')  
                else if S = X and X ∉ FV(T)  
                    then unify([X ↦ T]C') ∘ [X ↦ T]  
                else if T = X and X ∉ FV(S)  
                    then unify([X ↦ S]C') ∘ [X ↦ S]  
                else if S = S1 → S2 and T = T1 → T2  
                    then unify(C' ∪ {S1 = T1, S2 = T2})  
                else  
                    fail
```

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  $C = (\text{number of distinct type variables, total size of types})$ .

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

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



# Principle Types

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

$$\lambda x:X. \lambda y:Y. \lambda z:Z. (x z) (y z)$$

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

Unification Algorithm

**Theorem:** It is decidable whether  $(\Gamma, t)$  has a solution.



# Implicit Type Annotation

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

$$\frac{X \notin \mathcal{X} \quad \Gamma, x:X \vdash t_1 : T \quad |x \ C}{\Gamma \vdash \lambda x. t_1 : X \rightarrow T \quad |x \cup \{X\} \ C} \quad (\text{CT-ABSINF})$$



# Let-Polymorphism



- Code Duplication:

let doubleNat =  $\lambda f:\text{Nat} \rightarrow \text{Nat}. \lambda a:\text{Nat}. f(f(a))$  in

let doubleBool =  $\lambda f:\text{Bool} \rightarrow \text{Bool}. \lambda a:\text{Bool}. f(f(a))$  in

let a = doubleNat ( $\lambda x:\text{Nat}. \text{succ} (\text{succ } x)$ ) 1 in

let b = doubleBool ( $\lambda x:\text{Bool}. x$ ) false in ...Even



- One Attempt

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

let a = double ( $\lambda x:\text{Nat}. \text{succ} (\text{succ } x)$ ) 1 in

let b = double ( $\lambda x:\text{Bool}. x$ ) false in ...

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



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

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

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

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

Typable!



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

let  $x = \langle \text{utter garbage} \rangle$  in 5



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





- **Issue 2:** Avoid re-typechecking when a let-variable appear many times in **let  $x=t_1$  in  $t_2$** .

1. Find a principle type  $T_1$  of  $t_1$ .
2. Generalize  $T_1$  to a schema  $\forall X_1 \dots X_n. T_1$ .
3. Extend the context with  $(x, \forall X_1 \dots X_n. T_1)$ .
4. Each time we encounter an occurrence of  $x$  in  $t_2$ , look up its type scheme  $\forall X_1 \dots X_n. T_1$ , generate fresh type variables  $Y_1 \dots Y_n$  to instantiate the type scheme, yielding  $[X_1 \rightarrow Y_1, \dots, X_n \rightarrow Y_n]T_1$ , 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 type-checker 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>` :  $X \rightarrow X$

```
λz:ZZ. λy:YY. z (y true);
```

▶ `<fun>` :  $(?X_0 \rightarrow ?X_1) \rightarrow (\text{Bool} \rightarrow ?X_0) \rightarrow ?X_1$

```
λw:W. if true then false else w false;
```

▶ `<fun>` :  $(\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool}$

Type variables with names like `?X0` are automatically generated. □

