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

\[ X \]

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

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

\[ \text{dom}(\sigma) = \{X, Y\} \]
\[ \text{range}(\sigma) = \{\text{Bool}, \text{U}\} \]

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:

\[
\begin{align*}
\sigma(X) &= \begin{cases} 
T & \text{if } (X \rightarrow T) \in \sigma \\
X & \text{if } X \text{ is not in the domain of } \sigma
\end{cases} \\
\sigma(\text{Nat}) &= \text{Nat} \\
\sigma(\text{Bool}) &= \text{Bool} \\
\sigma(T_1 \rightarrow T_2) &= \sigma T_1 \rightarrow \sigma T_2
\end{align*}
\]

• 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:**
  - $\sigma(x_1:T_1,\ldots,x_n:T_n) = (x_1:\sigma T_1,\ldots,x_n:\sigma T_n)$.

• **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., \( \lambda f : X \to X. \lambda a : X. f (f a) \)

• **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 $\sigma \Gamma \vdash \sigma t : T$. 

\[ \begin{array}{c}
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (T\text{-VAR}) \\
\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . \ t_2 : T_1 \to T_2} \quad (T\text{-ABS}) \\
\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \quad (T\text{-APP})
\end{array} \]
Example: Let $\Gamma = f : X$, $a : Y$ and $t = f \ a$. Then

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

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

$$([X \to \text{Nat} \to \text{Nat}, Y \to \text{Nat}], \underline{\text{ }} \to \text{Nat})$$

are all solutions for $(\Gamma, t)$. 
Constraint-based Typing

The constraint typing relation
\[ \Gamma \vdash t : T \mid_X C \]
is defined as follows.

Exercise: Construct C from the term \( \lambda x:X, \lambda y:Y, \lambda z:Z.\ x \ z \ (y \ z) \)
• Extended with Boolean Expression

\[
\begin{align*}
\Gamma \vdash \text{true} : \text{Bool} & \mid \emptyset \ {\{\}} & \text{(CT-TRUE)} \\
\Gamma \vdash \text{false} : \text{Bool} & \mid \emptyset \ {\{\}} & \text{(CT-FALSE)} \\
\Gamma \vdash t_1 : T_1 & \mid x_1 \ C_1 \\
\Gamma \vdash t_2 : T_2 & \mid x_2 \ C_2 \\
\Gamma \vdash t_3 : T_3 & \mid x_3 \ C_3 \\
X_1, X_2, X_3 \text{ nonoverlapping} & \\
C' = C_1 \cup C_2 \cup C_3 \cup \{T_1 = \text{Bool}, T_2 = T_3\} & \\
\hline
\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 & \mid x_1 \cup x_2 \cup x_3 \ C' \\
\text{(CT-IF)} & 
\end{align*}
\]
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 $\text{dom}(\sigma) \cap X = \emptyset$,
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.
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

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

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.,
  \[ \lambda x: X. \lambda y: Y. \lambda z: Z. (x \ z) \ (y \ z) \]
  so that it becomes typable, then there is a most general or principal way of doing so.

**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.

\[
\frac{X \notin X \quad \Gamma, x : X \vdash t_1 : T \quad |_X \ C}{\Gamma \vdash \lambda x . t_1 : X \to T \quad |_{X \cup \{x\}} \ C} \quad \text{(CT-AbsINF)}
\]
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 ...
• One Attempt

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

This is not typable, since double can only be instantiated once.
• Solution: Unfolding “let” (perform a step of evaluation of let)

\[
\begin{align*}
\Gamma \vdash & \ [x \rightarrow t_1] t_2 : T_2 \\
\Gamma \vdash & \ \text{let } x = t_1 \ \text{in } t_2 : T_2
\end{align*}
\] (T-LetPoly)

\[
\begin{align*}
\Gamma \vdash & \ [x \rightarrow t_1] t_2 : T_2 \ \mid x \ C \\
\Gamma \vdash & \ \text{let } x = t_1 \ \text{in } t_2 : T_2 \ \mid x \ C
\end{align*}
\] (CT-LetPoly)

\[
\text{let double } = \lambda f. \lambda a. f(f(a)) \ \text{in}
\]

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

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

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

```
let x = <utter garbage> in 5
```

\[
\begin{align*}
\Gamma \vdash [x \mapsto t_1]t_2 : T_2 & \quad \Gamma \vdash t_1 : T_1 \\
\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2
\end{align*}
\] (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 $T_1$ of $t_1$.
2. Generalize $T_1$ to a schema $\forall X_1\ldots X_n.T_1$.
3. Extend the context with $(x, \forall X_1\ldots X_n.T_1)$.
4. Each time we encounter an occurrence of $x$ in $t_2$, look up its type scheme $\forall X_1\ldots X_n.T_1$, generate fresh type variables $Y_1\ldots Y_n$ to instantiate the type scheme, yielding $[X_1 \to Y_1, \ldots, X_n \to Y_n]T_1$, 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 typechecker that calculates principal types, taking the reconbase checker as a starting point. A typical interaction with your typechecker might look like:

\[ \lambda x : X. \ x; \]

- \( \text{<fun>} : X \rightarrow X \)

\[ \lambda z : ZZ. \lambda y : YY. \ z \ (y \ \text{true}); \]

- \( \text{<fun>} : (\ ?X_0 \rightarrow \ ?X_1) \rightarrow (\text{Bool} \rightarrow \ ?X_0) \rightarrow \ ?X_1 \)

\[ \lambda w : W. \ \text{if true then false else w false}; \]

- \( \text{<fun>} : (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \)

Type variables with names like \( ?X_0 \) are automatically generated.