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:

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

• Type substitution composition

\[
\sigma \circ \gamma = \left[ \begin{array}{ll}
X \rightarrow \sigma(T) & \text{for each } (X \rightarrow T) \in \gamma \\
X \rightarrow T & \text{for each } (X \rightarrow T) \in \sigma \text{ with } X \not\in \text{dom}(\gamma)
\end{array} \right]
\]
• 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., $\lambda f : T \to T. \lambda a : T. f (f a)$

• **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 $(\sigma, \tau)$ such that $\sigma \Gamma \vdash \sigma t : \tau$.
EXAMPLE: Let $\Gamma = f:X, a:Y$ and $t = f\ a$. Then

\[
\begin{align*}
([X \rightarrow Y \rightarrow \text{Nat}], \text{Nat}) & \quad ([X \rightarrow Y \rightarrow \text{Z}], \text{Z}) \\
([X \rightarrow Y \rightarrow \text{Z}, Z \rightarrow \text{Nat}], \text{Z}) & \quad ([X \rightarrow Y \rightarrow \text{Nat} \rightarrow \text{Nat}], \text{Nat} \rightarrow \text{Nat}) \\
([X \rightarrow \text{Nat} \rightarrow \text{Nat}, Y \rightarrow \text{Nat}], \text{Nat} \rightarrow \text{Nat})
\end{align*}
\]

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

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

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_\emptyset \{\}} \quad \text{(CT-VAR)}
\]

\[
\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid_\chi C}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2 \mid_\chi C} \quad \text{(CT-ABS)}
\]

\[\begin{align*}
\Gamma \vdash t_1 : T_1 & \mid_\chi_1 C_1 \\
\Gamma \vdash t_2 : T_2 & \mid_\chi_2 C_2 \\
\chi_1 \cap \chi_2 & = \chi_1 \cap FV(T_2) = \chi_2 \cap FV(T_1) = \emptyset \\
\chi & \not\subseteq \chi_1, \chi_2, T_1, T_2, C_1, C_2, \Gamma, t_1, \text{ or } t_2 \\
C' & = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}
\end{align*} \]

\[\Gamma \vdash t_1 \ t_2 : X \mid_{\chi_1 \cup \chi_2 \cup \{X\}} C' \quad \text{(CT-APP)}
\]
• Extended with Boolean Expression

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

• Case CT-Var.

\[ \frac{\text{\( x : T \in \Gamma \)}}{\Gamma \vdash x : T \mid \emptyset} \]  

(CT-VAR)

$(\sigma, \tau)$ is a solution $\Rightarrow \sigma T = \tau \Rightarrow \sigma \Gamma \vdash x : \tau$
• Case CT-Abs.

\[ \begin{array}{c}
\Gamma, x : T_1 \vdash t_2 : T_2 \mid x \ C \\
\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \mid x \ C
\end{array} \]  

(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

\[
\begin{align*}
\Gamma \vdash t_1 : T_1 \mid X_1 \ C_1 \quad \Gamma \vdash t_2 : T_2 \mid X_2 \ C_2 \\
X_1 \cap X_2 = X_1 \cap FV(T_2) = X_2 \cap FV(T_1) = \emptyset \\
X \not\in X_1, X_2, T_1, T_2, C_1, C_2, \Gamma, t_1, \text{or} \ t_2 \\
C' = C_1 \cup C_2 \cup \{T_1 = T_2 \rightarrow X\} \\
\Gamma \vdash t_1 \ t_2 : X \mid X_1 \cup X_2 \cup \{X\} \ C'
\end{align*}
\]

\[\text{(CT-APP)}\]

\((\sigma, \tau)\) is a solution \(\Rightarrow\) \(...\)
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 $\sigma \sqsubset \sigma'$, 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 $\sigma \sqsubset \sigma'$ 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

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

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.

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.

\[
\frac{X \notin X}{\Gamma, x : \Gamma \vdash t_1 : T \mid X \quad C} \quad (\text{CT-AbsINF})
\]

\[
\frac{\Gamma \vdash \lambda x. t_1 : X \rightarrow T \mid X \cup \{x\} \quad C}{\Gamma \vdash \lambda x. t_1 : X \rightarrow T \mid X \cup \{x\} \quad C}
\]
Let-Polymorphism

- Code Duplication:

let doubleNat = \ f: Nat → Nat. \ a: Nat. f(f(a)) in
let doubleBool = \ f: Bool → Bool. \ a: Bool. f(f(a)) in
let a = doubleNat (\ x: Nat. succ (succ x)) 1 in
let b = doubleBool (\ x: Bool. x) false in ...Even
• One Attempt

let double = \ f:X \rightarrow X. \ a:X. \ f(f(a)) \ in
let a = double (\ x:Nat. \ succ \ (succ \ x)) \ 1 \ in
let b = double (\ x: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)

\[
\frac{\Gamma \vdash [x \rightarrow t_1] t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}\]  \hspace{1cm} \text{(T-LetPoly)}

\[
\frac{\Gamma \vdash [x \rightarrow t_1] t_2 : T_2 \mid \lambda x \ C}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2 \mid \lambda x \ C}\]  \hspace{1cm} \text{(CT-LetPoly)}

let double = \lambda f. \lambda a. f(f(a)) in
let a = double (\lambda x:Nat. succ (succ x)) 1 in
let b = double (\lambda x:Bool. x) false in • Typable!
• **Issue 1**: what happens when the let-bound variable does not appear in the body:

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

\[
\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \quad \Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad (T\text{-LETPOLY})
\]
• **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 $\forall X_1...X_n.T_1$.
3. Extend the context with $(x, \forall X_1...X_n.T_1)$.
4. Each time we encounter an occurrence of $x$ in $t_2$, look up its type scheme $\forall X_1...X_n.T_1$, generate fresh type variables $Y_1...Y_n$ to instantiate the type scheme, yielding $[X_1 \rightarrow Y_1, \ldots, X_n \rightarrow Y_n]T_1$, 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:

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

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

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

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

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

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

Type variables with names like ?X_0 are automatically generated.