Lecture 6: Type Reconstruction

- Type Variables and Substitutions
- Two View of Type Variables
- Constraint-Based Typing
- Unification
- Principal Types
- Let Polymorphism
Type Variables and Substitutions

In this lecture, we treat uninterpreted base types as type variables.

A type X can stand for Nat → Bool. We may need to substitute X by the desired type Nat → Bool.

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

$$\sigma = [X \mapsto T, Y \mapsto U]$$

where

$$\text{dom}(\sigma) = \{X, Y\}$$
$$\text{range}(\sigma) = \{T, U\}$$
Applying Substitutions to Types

\[ \sigma (X) = \begin{cases} T & \text{if } (X \rightarrow T) \in \sigma \\ X & \text{if } X \notin \text{dom}(\sigma) \end{cases} \]

\[ \sigma (\text{Nat}) = \text{Nat} \]

\[ \sigma (\text{Bool}) = \text{Bool} \]

\[ \sigma (T_1 \rightarrow T_2) = \sigma T_1 \rightarrow \sigma T_2 \]
Applying Substitutions to Contexts/Terms

Applying it to contexts:

\[ \sigma (x_1:T_1, \ldots, x_n:T_n) = (x_1: \sigma T_1, \ldots, x_n: \sigma T_n) \]

Applying it to terms by applying it to all its types. E.g:

\[ [X \mapsto \text{Bool}] (\lambda x:X. x) = \lambda x:\text{Bool}. x \]
Apply $\gamma$ followed by $\sigma$, as follows:

\[
\sigma \circ \gamma = X \mapsto \sigma(T) \text{ for each } (X \mapsto T) \in \gamma
\]

\[
= X \mapsto T \text{ for each } (X \mapsto T) \in \sigma \text{ with } X \notin \text{dom}(\gamma)
\]
Preservation under Type Substitution

If \( \Gamma \vdash t : T \)

then \( \sigma \Gamma \vdash \sigma t : \sigma T \)

for any type substitution \( \sigma \)
First View of Type Equation Solving

Let $t$ be a term with type variables, and let $\Gamma$ be a typing context with type variables.

First View:
For every $\sigma$ there exists a $T$ such that $\sigma \Gamma \vdash \sigma t : \sigma T$.

“Are all substitution instances of $t$ well-typed?”

This view leads to parametric polymorphism.
Second View of Type Equation Solving

Let \( t \) be a term with type variables, and let \( \Gamma \) be a typing context with type variables.

Second View:

Is there a \( \sigma \) such that there is a \( T \) whereby

\[
\sigma \Gamma \vdash \sigma t : \sigma T.
\]

“Is some substitution instance of \( t \) well-typed?”

This view leads to type reconstruction.
Type Reconstruction: The Problem

Let \( t \) be a term and \( \Gamma \) be a typing context.

A solution for \((\Gamma, t)\) is a pair \((\sigma, T)\) such that \(\sigma \Gamma \vdash \sigma t : \sigma T\)
Example

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

Then the possible solutions for $(\Gamma, t)$ include:

- $([X \mapsto Y \to \text{Nat}], \text{Nat})$
- $([X \mapsto Y \to Z], \text{Z})$
- $([X \mapsto Y \to Z, Z \mapsto \text{Nat }], \text{Z})$
- $([X \mapsto Y \to \text{Nat} \to \text{Nat}], \text{Nat} \to \text{Nat})$
- $([X \mapsto \text{Nat} \to \text{Nat}, Y \mapsto \text{Nat }], \text{Nat})$
**Constraint-based Typing**

Constraint-based typing is an algorithm that computes for \((\Gamma, t)\) a set of *constraints* that must be satisfied by any solution for \((\Gamma, t)\).

A *constraint* set \(C\) is a set of solutions \(\{S_i = T_i\}_{i \in 1..n}\). A substitution \(\sigma\) *unifies* an equation \(S = T\) if \(\sigma S\) and \(\sigma T\) are *identical*, namely \(\sigma S \equiv \sigma T\).

A substitution *unifies* (or *satisfies*) a constraint set \(C\) if it unifies every equation in \(C\).
**Constraint-based Typing**

We define a relation

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

The term $t$ has type $T$ under assumptions $\Gamma$ whenever the constraint $C$ are satisfied.

$X$ is used to track variables that are introduced along the way.
Rules for Constraint-Based Typing

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

\[
\Gamma \vdash 0 : \text{Nat} \mid \emptyset \{\} \quad \text{(CT-Zero)}
\]

\[
\frac{\Gamma \vdash t : T \mid_x C \quad C' = C \cup \{T=\text{Nat}\}}{\Gamma \vdash \text{succ } t : \text{Nat} \mid_x C'} \quad \text{(CT-Succ)}
\]

\[
\frac{\Gamma \vdash t : T \mid_x C \quad C' = C \cup \{T=\text{Nat}\}}{\Gamma \vdash \text{pred } t : \text{Nat} \mid_x C'} \quad \text{(CT-Pred)}
\]
Rules for Constraint-Based Typing

\[ \Gamma \vdash \text{true} : \text{Bool} \mid \emptyset \{ \} \]  
(CT-True)

\[ \Gamma \vdash \text{false} : \text{Bool} \mid \emptyset \{ \} \]  
(CT-False)

\[ \frac{\Gamma \vdash t : T \mid \chi C \quad C' = C \cup \{ T = \text{Nat} \}}{\Gamma \vdash \text{iszero } t : \text{Bool} \mid \chi C'} \]  
(CT-IsZero)

\[ \frac{\Gamma \vdash t_1 : T_1 \mid \chi_1 C_1 \quad \Gamma \vdash t_2 : T_2 \mid \chi_2 C_2 \quad \Gamma \vdash t_3 : T_3 \mid \chi_3 C_3 \quad C' = C_1 \cup C_2 \cup C_3 \cup \{ T_1 = \text{Bool}, T_2 = T_3 \} \quad X' = X_1 \cup X_2 \cup X_3}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \mid \chi \cdot C'} \]  
(CT-If)
Rules for Constraint-Based Typing

\[ \Gamma, x: T_1 \vdash t_2 : T_2 \mid_x C \]
\[ \Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \mid_x C \]  

\[ \Gamma \vdash t_1 : T_1 \mid_{x_1} C_1 \quad \Gamma \vdash t_2 : T_2 \mid_{x_2} C_2 \]

\[ \text{fresh } V \quad C' = C_1 \cup C_2 \cup \{ T_1 = T_2 \rightarrow V \} \]
\[ X' = X_1 \cup X_2 \cup \{ V \} \]
\[ \Gamma \vdash t_1 t_2 : V \mid_{x'} C' \]  

(CT-Abs)

(CT-App)

Note that \( X_1, X_2, \text{FV}(T_2), \text{FV}(T_1) \) are disjoint.
**Constraint-based Typing (Solution)**

Suppose that

\[ \Gamma \vdash t : T \mid_X C \]

A solution for \((\Gamma, t, S, C)\) is a pair \((\sigma, T)\) such that \(\sigma\) satisfies \(C\) and \(\sigma S = T\).

Note that it is OK to omit \(X\) from discussion as it is simply a set of locally introduced type variables.
Properties of Constraint-based Typing

Soundness:

Suppose that $\Gamma \vdash t : T \mid_X C$. If $(\sigma, T)$ is a solution for $(\Gamma, t, S, C)$, then it is also a solution for $(\Gamma, t)$. That is $\sigma \Gamma \vdash \sigma t : \sigma T$.

Completeness:

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

Note that $\sigma \setminus X$ is a substitution that is undefined for all variables in $X$, but otherwise behaves like $\sigma$. 
Correctness of Constraint-based Typing

Suppose $\Gamma \vdash t : T \mid x \ C$.

There is some solution for $(\Gamma, t)$ if and only if there is some solution for $(\Gamma, t, S, C)$.

Correctness = Soundness + Completeness
**More General Substitution**

A substitution $\sigma$ is *more general* (or *less specific*) than a substitution $\sigma'$, written as $\sigma \sqsubseteq \sigma'$, if $\sigma' = \gamma \circ \sigma$ for some substitution $\gamma$.

For example:

$[X \mapsto V \rightarrow V, \ Y \mapsto W \rightarrow W]$ is *less specific* than $[X \mapsto (\text{Nat} \rightarrow \text{Nat}) \rightarrow [(\text{Nat} \rightarrow \text{Nat}) \ , \ Y \mapsto \text{Nat} \rightarrow \text{Nat}]]$

Take $\gamma = [V \mapsto \text{Nat} \rightarrow \text{Nat}, W \mapsto \text{Nat} ]$. 
A principal unifier for a constraint set $C$ is a substitution $\sigma$ such that:

- $\sigma$ satisfies $C$, and
- for every $\sigma'$ that satisfies $C$, we have $\sigma \subseteq \sigma'$.

That is,

$\sigma$ is the most general substitution that satisfies $C$. 
Examples

What is the principal unifier of the following?

\{X = \text{Nat}, Y = X \rightarrow X\}

\Rightarrow [X \mapsto \text{Nat}, Y \mapsto \text{Nat} \rightarrow \text{Nat}]

\{X \rightarrow Y = Y \rightarrow Z, Z = U \rightarrow W\}

\Rightarrow [X \mapsto U \rightarrow W, Y \mapsto U \rightarrow W, Z \mapsto U \rightarrow W]
Unification Algorithm

This derives principal unifier from a set of constraints.

\[
\text{unify}(C) = \begin{cases} 
\text{if } C = \{\} \text{ then } [] \\
\text{else let } \{S=T\} \cup C' = C \text{ in} \\
\quad \text{if } S \equiv T \text{ then } \text{unify}(C') \\
\quad \text{else if } S \equiv X \land X \not\in \text{FV}(T) \text{ then } \text{unify([X} \mapsto \text{T}]C') \circ [X \mapsto \text{T}] \\
\quad \text{else if } T \equiv X \land X \not\in \text{FV}(S) \text{ then } \text{unify([X} \mapsto \text{S}]C') \circ [X \mapsto \text{S}] \\
\quad \text{else if } S \equiv S_1 \rightarrow S_2 \land T \equiv T_1 \rightarrow T_2 \text{ then } \text{unify}(C' \cup \{S_1=\text{T}_1, S_2=\text{T}_2\}) \\
\quad \text{else } \text{fail}
\end{cases}
\]
Unification Algorithm (Properties)

Let $C$ be an arbitrary constraint set.

- $\text{unify}(C)$ terminates, either with fail or by returning a substitution.
- If $\text{unify}(C) = \sigma$ then $\sigma$ is a unifier for $C$.
- If $\delta$ is a unifier for $C$, then $\text{unify}(C) = \sigma$ for some $\sigma$ such that $\sigma \subseteq \delta$. 
**Principal Types**

A *principal solution* for \((\Gamma, t, S, C)\), is a solution \((\sigma, T)\), such that, whenever \((\sigma', T')\) is a solution for \((\Gamma, t, S, C)\), we have \(\sigma \subseteq \sigma'\).

When \((\sigma, T)\) is a principal solution, we call \(T\) a principal type for \(t\) under \(\Gamma\).
Unification Finds Principal Solution

If \((\Gamma,t,S,C)\) has any solution, then it has a principal one.

The unification algorithm can be used to determine whether \((\Gamma,t,S,C)\) has a solution and, if so, to calculate a principal solution.
Let-Polymorphism (Motivation)

Consider a function that applies the first argument twice to the second argument:

\[ \lambda f. \lambda a. f(f(a)) \]

This function has few assumptions on \( f \) and \( a \).

Can we apply the function, whenever these conditions are met?
Let-Polymorphism (Example)

We can use let construct to capture more generic code:

```
let double = \ f. \ a. f(f(a)) in
  ... double (\ x. succ(succ(x))) 1 ...
  ... double (\ x. not(x)) false ...
```

However, what type should `double` have?
Let-Polymorphism (Initial Idea)

Provide type variable for double:

\[
\text{let double } = \lambda f : X \rightarrow X. \lambda a : X. f(f(a)) \text{ in} \\
\quad \ldots \text{ double } (\lambda x. \text{succ}(\text{succ}(x))) \ 1 \ldots \\
\quad \ldots \text{ double } (\lambda x. \text{not}(x)) \text{ false} \ldots
\]

However, the let typing rule:

\[
\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2
\]

\[
\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2
\]

\(\text{(T-Let)}\)

generates the following \textit{contradiction}!

\[
X \rightarrow X = \text{Nat} \rightarrow \text{Nat} \\
X \rightarrow X = \text{Bool} \rightarrow \text{Bool}
\]
Let-Polymorphism (Second Idea)

Use implicitly annotated lambda abstraction:

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

\[
\text{... double } (\lambda \, x: \text{Nat}. \, \text{succ(succ(x))) } \, 1 \, ...
\]

\[
\text{... double } (\lambda \, x: \text{Bool}. \, \text{not(x)) } \, \text{false } ...
\]

Typing rule substitute all occurrences of double in body:

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

Problems

(i) what if \( x \) not used in \( t_2 \)

(ii) what if \( x \) occurs multiple times
Let-Polymorphism (Problem 1)

What if $x$ is not used in $t_2$?

Modify the type rule:

$$
\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash [x \mapsto t_1]t_2 : T_2
\quad \overline{\quad \Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}
$$
Let-Polymorphism *(Problem 2)*

What if \( x \) occurs multiple times?

Explicit substitution of each occurrence of variable may result in slow type-checking.

**Solution**: use *type schemes*. Resulting implementations of type reconstruction run in *practice in linear time*.

In theory, they are exponential as shown by Kfoury, Tiuryn and Urzyczyn (1990) since types can be exponential in size to program!
Let-polymorphism does not work correctly with references:

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

This results in run-time error even though it type-checks. Reason - mismatch between *evaluation rule* and *type rule*.

**Solution**: use polymorphism only if the RHS of let is a *value*.
**Unification Algorithm (Background)**

- Unification is due to J Alan Robinson (1971), and is widely used in computer science.

- Logic programming is based on unification over first-order terms. It is a generalization of our language of types. Unification is built-in.

- Occurs check is justified because we consider only finite types (ie. non-recursive types).