CS6202: Advanced Topics in Programming Languages and Systems

Lecture 6 : **Type Reconstruction**

- Type Variables and Susbtitutions
- 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

$$\mathrm{dom}(\sigma) = \{X, Y\}$$
$$\mathrm{range}(\sigma) = \{T, U\}$$

Applying Substitutions to Types

$$\begin{aligned}
\sigma(X) &= T \text{ if } (X \mapsto T) \in \sigma \\
&= X \text{ if } X \notin \mathrm{dom}(\sigma) \\
\sigma(\mathrm{Nat}) &= \mathrm{Nat} \\
\sigma(\mathrm{Bool}) &= \mathrm{Bool} \\
\sigma(T_1 \to T_2) &= \sigma T_1 \to \sigma T_2
\end{aligned}$$

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 :

$$[X \mapsto \mathrm{Bool}]\ (\lambda\ x{:}X.\ x) = \lambda\ x{:}\mathrm{Bool}.\ x$$

Composing Substitutions

Apply γ followed by σ, as follows:

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

$$\quad = \quad X \mapsto T \text{ for each } (X \mapsto T) \in \sigma \text{ with } X \notin dom(\gamma)$$

Preservation under Type Substitution

If $\Gamma \vdash t : T$

then $\sigma\Gamma \vdash \sigma t : \sigma T$
for any type substitution σ

First View of Type Equation Solving

Let t be a term with type variables, and let Γ be a typing context with type variables.

First View:
For every σ 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 Γ be a typing context with type variables.

Second View:
Is there a σ 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 Γ be a typing context.

A solution for (Γ, t) is a pair (σ, 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 (Γ, t) include:

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

Constraint-based Typing

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

A *constraint* set C is a set of solutions $\{S_i{=}T_i\}^{i \in 1..n}$. A substitution σ *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

$$\Gamma \vdash t : T \;|_X\, C$$

The term t has type T under assumptions Γ whenever the constraint C are satisfied.

X is used to track variables that are introduced along the way.

Rules for Constraint-Based Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \;|_{\varnothing}\; \{\}} \quad \text{(CT-Var)}$$

$$\Gamma \vdash 0 : Nat \;|_{\varnothing}\; \{\} \quad \text{(CT-Zero)}$$

$$\frac{\Gamma \vdash t : T \;|_X\; C \qquad C' = C \cup \{T = Nat\}}{\Gamma \vdash \text{succ } t : Nat \;|_X\; C'} \quad \text{(CT-Succ)}$$

$$\frac{\Gamma \vdash t : T \;|_X\; C \qquad C' = C \cup \{T = Nat\}}{\Gamma \vdash \text{pred } t : Nat \;|_X\; C'} \quad \text{(CT-Pred)}$$

Rules for Constraint-Based Typing

$$\Gamma \vdash \text{true} : Bool \;|_{\varnothing}\; \{\} \quad \text{(CT-True)}$$

$$\Gamma \vdash \text{false} : Bool \;|_{\varnothing}\; \{\} \quad \text{(CT-False)}$$

$$\frac{\Gamma \vdash t : T \;|_X\; C \qquad C' = C \cup \{T = Nat\}}{\Gamma \vdash \text{iszero } t : Bool \;|_X\; C'} \quad \text{(CT-IsZero)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \;|_{X1}\; C_1 \quad \Gamma \vdash t_2 : T_2 \;|_{X2}\; C_2 \quad \Gamma \vdash t_3 : T_3 \;|_{X3}\; C_3 \\ C' = C_1 \cup C_2 \cup C_3 \cup \{T_1 = Bool, T_2 = T_3\} \\ X' = X1 \cup X2 \cup X3\end{array}}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \;|_{X'}\; C'} \quad \text{(CT-If)}$$

Rules for Constraint-Based Typing

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2 \;|_X\; C}{\Gamma \vdash \lambda x{:}T_1 .\, t_2 : T_1 \to T_2 \;|_X\; C} \quad \text{(CT-Abs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \;|_{X1}\; C_1 \qquad \Gamma \vdash t_2 : T_2 \;|_{X2}\; C_2 \\ \text{fresh } V \qquad C' = C_1 \cup C_2 \cup \{T_1 = T_2 \to V\} \\ X' = X1 \cup X2 \cup \{V\}\end{array}}{\Gamma \vdash t_1\, t_2 : V \;|_{X'}\; C'} \quad \text{(CT-App)}$$

Note that $X1, X2, FV(T_2), FV(T_1)$ are disjoint.

Constraint-based Typing (Solution)

Suppose that

$$\Gamma \vdash t : T \;|_X\; C$$

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

Note that is 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 (σ,T) is a solution for (Γ,t,S,C), then it is also a solution for (Γ,t). That is $\sigma\Gamma \vdash \sigma t : \sigma T$.

Completeness:

Suppose that $\Gamma \vdash t : T \mid_X C$. If (σ,T) is a solution for (Γ,t) and $dom(\sigma) \cap X=\{\}$, then there is a solution (σ',T) for (Γ,t,S,C) such that $\sigma' \backslash X = \sigma$.
Note that $\sigma \backslash X$ is a substitution that is undefined for all variables in X, but otherwise behaves like σ.

Correctness of Constraint-based Typing

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

There is some solution for (Γ,t) *if and only if* there is some solution for (Γ,t,S,C) .

Correctness = Soundness + Completeness

More General Substitution

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

For example:

$[X \mapsto V \to V, Y \mapsto W \to W]$ is *less specific* than
$[X \mapsto (Nat \to Nat) \to [(Nat \to Nat) , Y \mapsto Nat \to Nat]$

Take $\gamma = [V \mapsto Nat \to Nat, W \mapsto Nat]$.

Principal Unifier

A *principal unifier* for a constraint set C is a substitution σ such that:

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

That is,
σ is the *most general* substitution that satisfies C.

Examples

What is the principal unifier of the following?

$\{X=Nat, Y= X \to X\}$

$\boxed{\Rightarrow [X \mapsto Nat, Y \mapsto Nat \to Nat]}$

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

$\boxed{\Rightarrow [X \mapsto U \to W, Y \mapsto U \to W , Z \mapsto U \to W]}$

Unification Algorithm

This derives principal unifier from a set of constraint

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

occurs check

Unification Algorithm (Properties)

Let C be an arbitrary constraint set.

- unify(C) terminates, either with fail or by returning a substitution.
- If unify(C)=σ then σ is a unifier for C.
- If δ is a unifier for C, then unify(C)=σ for some σ such that $\sigma \sqsubseteq \delta$.

Principal Types

A *principal solution* for (Γ,t,S,C), is a solution (σ,T), such that, whenever (σ',T') is a solution for (Γ,t,S,C), we have $\sigma \sqsubseteq \sigma'$.

When (σ,T) is a principal solution, we call T a principal type for t under Γ.

Unification Finds Principal Solution

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

The unification algorithm can be used to determine whether (Γ,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:

λ f. λ 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 = λ f. λ a. f(f(a)) in
… double (λ x. succ(succ(x))) 1 …
… double (λ x. not(x)) false …

However, what type should double have?

Let-Polymorphism (Initial Idea)

Provide type variable for double:

let double = λ f : X → X. λ a:X. f(f(a)) in
… double (λ x. succ(succ(x))) 1 …
… double (λ x. not(x)) false …

However, the let typing rule :

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x{=}t_1 \text{ in } t_2 : T_2} \quad \text{(T-Let)}$$

generates the following *contradiction*!

$$X \to X = Nat \to Nat$$
$$X \to X = Bool \to Bool$$

Let-Polymorphism (Second Idea)

Use implicitly annotated lambda abstraction:

let double = λ f . λ a. f(f(a)) in
… double (λ x:Nat. succ(succ(x))) 1 …
… double (λ x:Bool. not(x)) 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:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \text{let } x{=}t_1 \text{ in } t_2 : T_2}$$

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!

Problem with References

Let-polymorphism does not work correctly with references:

let r=ref (λ x.x) in
r:=(λ 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).