

# 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

CS6202

Lecture 7 : Type Reconstruction

### **Applying Substitutions to Types**

 $\begin{aligned} \sigma \left( X \right) & = T \text{ if } \left( X \mapsto T \right) \in \sigma \\ & = X \text{ if } X \notin dom(\sigma) \end{aligned}$ 

- $\sigma$  (Nat) = Nat
- $\sigma$  (Bool) = Bool

$$\boldsymbol{\sigma} \left( \boldsymbol{T}_{1} \rightarrow \boldsymbol{T}_{2} \right) \ = \ \boldsymbol{\sigma} \ \boldsymbol{T}_{1} \rightarrow \ \boldsymbol{\sigma} \ \boldsymbol{T}_{2}$$

# **Type Variables and Substitutions**

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

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

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

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

where

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

CS6202

Lecture 7 : Type Reconstruction

2

### **Applying Substitutions to Contexts/Terms**

Applying it to contexts:

 $\boldsymbol{\sigma} (\mathbf{x}_1:\mathbf{T}_1,\ldots,\mathbf{x}_n:\mathbf{T}_n) = (\mathbf{x}_1:\boldsymbol{\sigma} \mathbf{T}_1,\ldots,\mathbf{x}_n:\boldsymbol{\sigma} \mathbf{T}_n)$ 

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

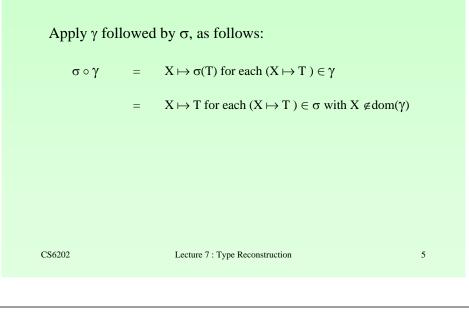
 $[X\mapsto Bool]\ (\lambda \ x{:}X{.}\ x)\ =\ \lambda \ x{:}Bool{.}\ x$ 

CS6202

3

1

### **Composing Substitutions**



### Preservation under Type Substitution

If	$\Gamma \vdash t:T$	
then	σΓ⊢σt:σΤ for any type substitution σ	
CS6202	Lecture 7 : Type Reconstruction	6

### 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.

7

CS6202

### **Type Reconstruction : The Problem**

Let t be a term and  $\Gamma$  be a typing context.

### Example

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

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

 $\begin{array}{ll} ([X \mapsto Y \to Nat], & Nat) \\ ([X \mapsto Y \to Z], & Z) \\ ([X \mapsto Y \to Z, Z \mapsto Nat], & Z) \\ ([X \mapsto Y \to Nat \to Nat], & Nat \to Nat) \\ ([X \mapsto Nat \to Nat, Y \mapsto Nat], & Nat) \end{array}$ 

CS6202	Lecture 7 : Type Reconstruction	9	CS6202	Lecture 7 : Type Reconstruction	10

### **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.

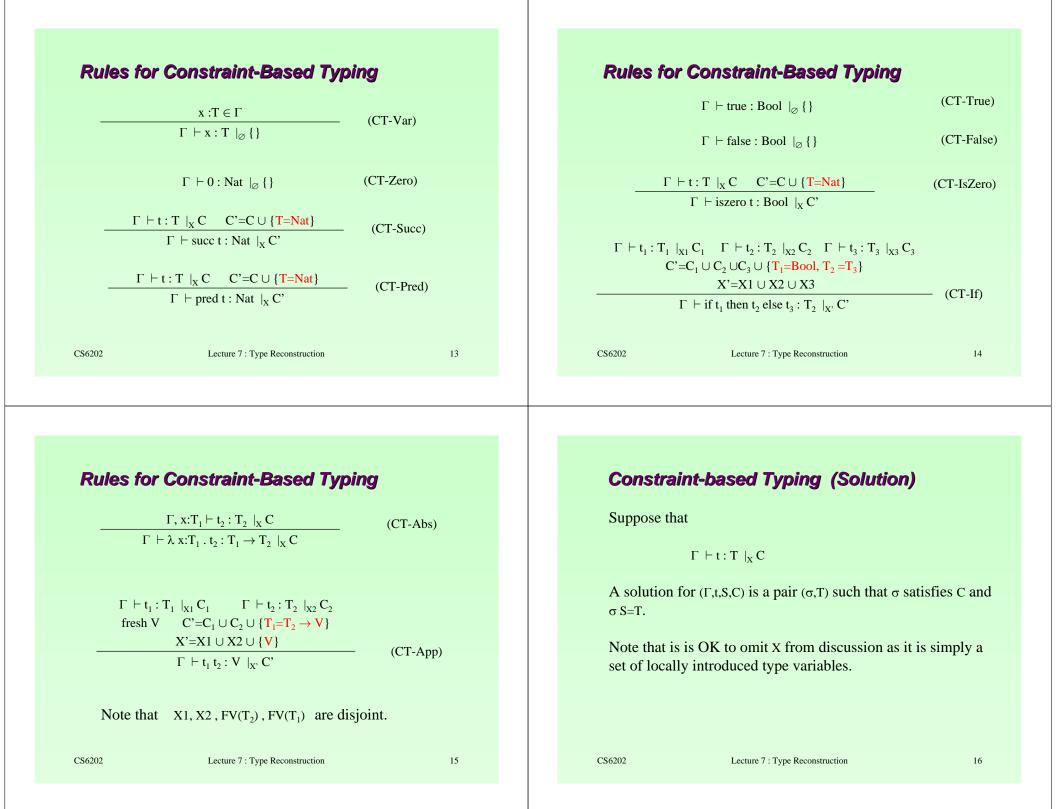
### **Constraint-based Typing**

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.



### **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  $dom(\sigma) \cap X = \{\}$ , 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  $\sigma$ .

CS6202	Lecture 7 : Type Reconstruction	17	CS6202	Lecture 7 : Type Reconstruction

### **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:

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

```
Take \gamma = [V \mapsto Nat \rightarrow Nat, W \mapsto Nat].
```

### **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

Principal Unifier

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  $\sigma \sqsubseteq \sigma$ '.

#### That is,

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

19

18

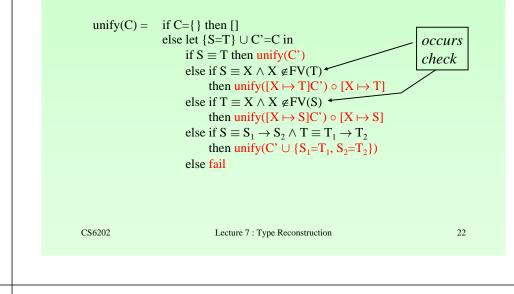
### **Examples**

What is the principal unifier of the following?

 $\{X=Nat, Y=X \to X\}$   $\Rightarrow [X \mapsto Nat, Y \mapsto Nat \to Nat]$   $\{X \to Y=Y \to Z, Z=U \to W\}$   $\Rightarrow [X \mapsto U \to W, Y \mapsto U \to W, Z \mapsto U \to W]$ CS6202
Lecture 7: Type Reconstruction

### **Unification Algorithm**

This derives principal unifier from a set of constraint



### **Unification Algorithm (Properties)**

Let C be an arbitrary constraint set.

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

### **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  $\sigma \sqsubseteq \sigma$ '.

When  $(\sigma,T)$  is a principal solution, we call T a principal type for t under  $\Gamma.$ 

21

### **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.

CS6202	Lecture 7 : Type Reconstruction	25	

### 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?

Lecture 7 : Type Reconstruction

### Let-Polymorphism (Example)

We can use let construct to capture more generic code:

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

However, what type should double have?

### Let-Polymorphism (Initial Idea)

```
Provide type variable for double:

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

... double (\lambda x. succ(succ(x))) 1 ...

... double (\lambda x. not(x)) false ...
```

However, the let typing rule :

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

generates the following contradiction!

 $\begin{array}{ll} X 
ightarrow X \;=\; Nat 
ightarrow Nat \ X 
ightarrow X \;=\; Bool 
ightarrow Bool \end{array}$ 

27

Lecture 7 : Type Reconstruction

26

CS6202

### Let-Polymorphism (Second Idea) Let-Polymorphism (Problem 1) Use implicitly annotated lambda abstraction: let double = $\lambda$ f . $\lambda$ a. f(f(a)) in What if x is not used in $t_2$ ? ... double ( $\lambda$ x:Nat. succ(succ(x))) 1 ... ... double ( $\lambda$ x:Bool. not(x)) false ... Modify the type rule: Typing rule substitute all occurrences of double in body: $\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash [\mathbf{x} \mapsto t_1]t_2 : T_2}{\Gamma \vdash \text{let } \mathbf{x} = t_1 \text{ in } t_2 : T_2}$ $\Gamma \vdash [\mathbf{x} \mapsto \mathbf{t}_1]\mathbf{t}_2 : \mathbf{T}_2$ (T-LetPoly) $\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : T_2$ **Problems** (i) what if x not used in $t_2$ (ii) what if x occurs multiple times CS6202 Lecture 7 : Type Reconstruction CS6202 Lecture 7 : Type Reconstruction 30 29

### Let-Polymorphism (Problem 2)

What if x occurs multiple times?

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

<u>Solution</u> : 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 ( $\lambda$  x.x) in r:=( $\lambda$  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*.

31

CS6202

## 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).

CS6202	Lecture 7 : Type Reconstruction	33