Lund University
Computer Science Department

Type Reconstruction

TAPL 2022

Idriss Riouak

January 25, 2022

Ch. 22 - Type reconstruction

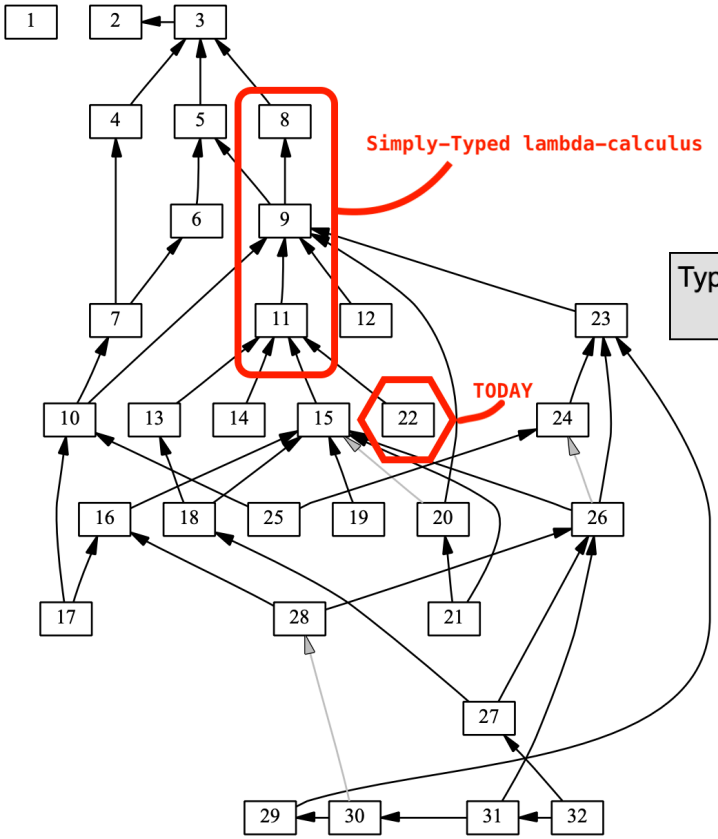
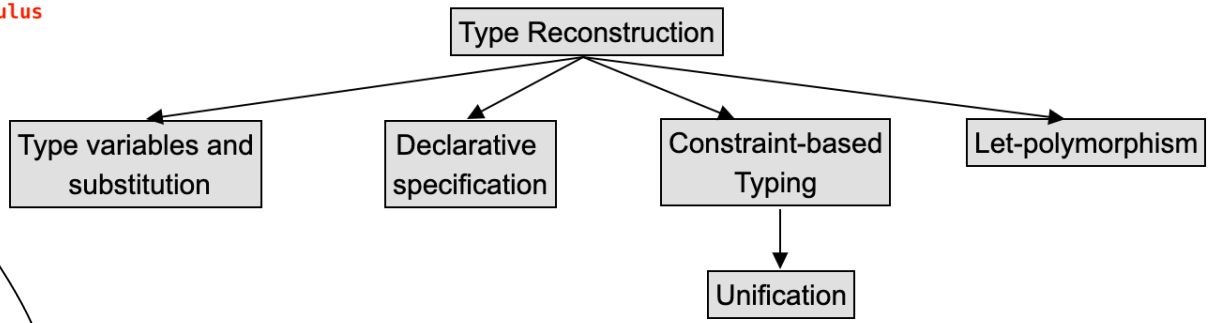


Figure P-1: Chapter dependencies



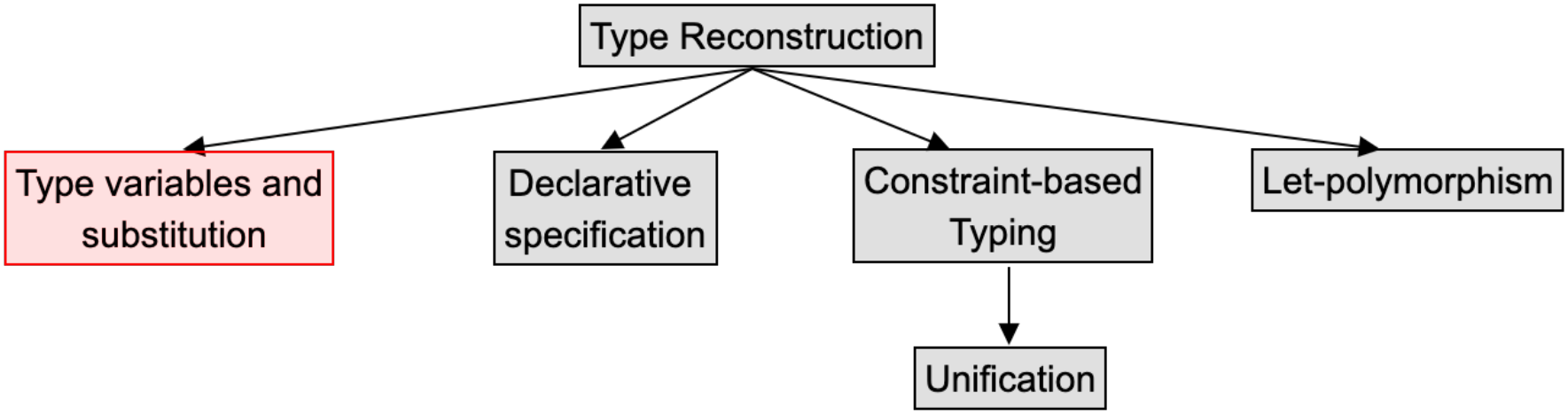
Goal of today lecture

Type checking: Given Γ, t and T, check whether $\Gamma \models t : T$.

Type reconstruction: Given Γ and t, find a type T s.t. $\Gamma \models t : T$.

LTH
FACULTY OF ENGINEERING

Outline


Type Reconstruction
Type variables and substitution
Declarative specification
Constraint-based Typing
Let-polymorphism
Unification

Operations on type variables

Let us consider the following term:

$$\lambda g : Y.\lambda a : X.g(g\ a)$$

it is not typable as it stands.

Operations on type variables

Let us consider the following term:

$$\lambda g : Y.\lambda a : X.g(g\ a)$$

it is not typable as it stands. It is typable if we **substitute** Y with $\texttt{Nat} \to \texttt{Nat}$ and X with $\texttt{Nat}$:

$$\lambda g : \texttt{Nat} \to \texttt{Nat}.\lambda a : \texttt{Nat}.g(g\ a)$$

Operations on type variables

Let us consider the following term:

$$\lambda g : Y.\lambda a : X.g(g\ a)$$

it is not typable as it stands. It is typable if we **substitute** Y with $\texttt{Nat} \to \texttt{Nat}$ and X with $\texttt{Nat}$:

$$\lambda g : \texttt{Nat} \to \texttt{Nat}.\lambda a : \texttt{Nat}.g(g\ a)$$

Definition

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

LTH
FACULTY OF ENGINEERING

Operations on type variables

Let us consider the following term:

$$\lambda g : Y.\lambda a : X.g(g\ a)$$

it is not typable as it stands. It is typable if we **substitute** Y with $\texttt{Nat} \to \texttt{Nat}$ and X with $\texttt{Nat}$:

$$\lambda g : \texttt{Nat} \to \texttt{Nat}.\lambda a : \texttt{Nat}.g(g\ a)$$

Definition

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

Example

The substitution $[Y \mapsto X \to X, X \mapsto \texttt{Bool}]$ will map X to $\texttt{Bool}$, and Y to $X \to X$, not $\texttt{Bool} \to \texttt{Bool}$

LTH FACULTY OF ENGINEERING

Operation on type variables

Application of a substitution σ to a type:

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

$$\sigma(Nat) = Nat$$

$$\sigma(Bool) = Bool$$

$$\sigma(T_1 \to T_2) = \sigma(T_1) \to \sigma(T_2)$$

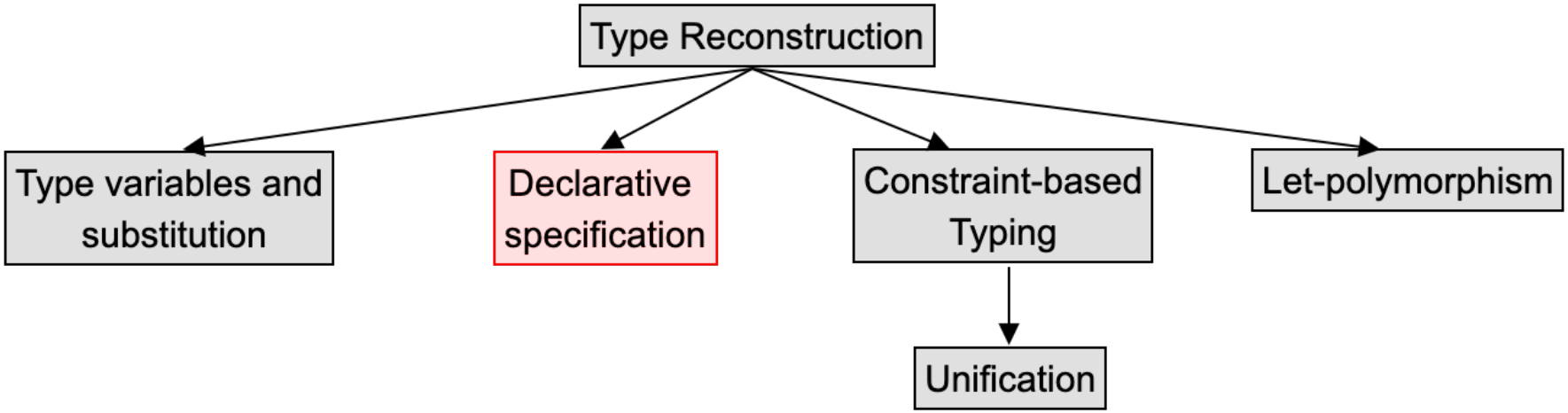
If σ and γ are substitutions:

$$\sigma \circ \gamma = \begin{cases} X \mapsto \sigma(T) & \text{for each } (X \mapsto T) \in \gamma \\ X \mapsto T & \text{for each } (X \mapsto T) \in \sigma \wedge X \notin dom(\gamma) \end{cases}$$

TL;DR: $(\sigma \circ \gamma)S = \sigma(\gamma S)$

LTH
FACULTY OF ENGINEERING

Outline


Type Reconstruction
Type variables and substitution
Declarative specification
Constraint-based Typing
Let-polymorphism
Unification

Two interesting questions

- "Are all substitution instances of t well typed ?"
  i.e., $\forall\sigma.\sigma\Gamma \models \sigma t : T$ for some T ?

Example

$\lambda g : X \to X.\lambda a : X.g(g\ a)$

- "Is **some** substitution instance of t well typed" ?
  i.e., $\exists\sigma : \sigma\Gamma \models \sigma t : T$ for some T ?

Example

$\lambda g : Y \to X.\lambda a : X.g(g\ a)$

Looking for **valid instantiations** of type variables leads to the ideas of type *reconstruction* (or type inference). The programmer (as in ML or Haskell) may leave out all type annotations.

LTH
FACULTY OF ENGINEERING

Type solution

Definition

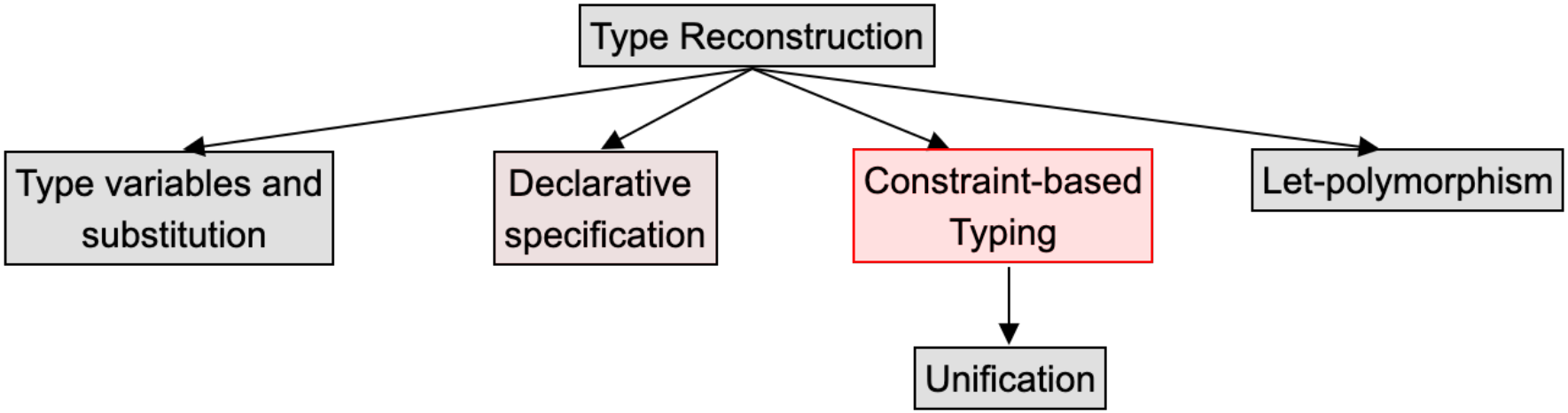
Let Γ be a context and t a term. A **solution** for (Γ, t) is a pair $(\sigma, \mathbf{T})$ s.t. $\sigma\Gamma \models \sigma t : \mathbf{T}$.

Example

Let $\Gamma = g : X, a : Y$ and $t = g\ a$. Then all the solutions for (Γ, t) are:

- $([X \mapsto Y \to \mathtt{Nat}], \mathtt{Nat})$
- $([X \mapsto Y \to Z, Z \mapsto \mathtt{Nat}], Z)$
- $([X \mapsto \mathtt{Nat} \to \mathtt{Nat}, Y \mapsto \mathtt{Nat}], \mathtt{Nat})$
- $([X \mapsto Y \to Z], Z)$
- $([X \mapsto Y \to \mathtt{Nat} \to \mathtt{Nat}], \mathtt{Nat})$

Outline


Type Reconstruction
Type variables and substitution
Declarative specification
Constraint-based Typing
Let-polymorphism
Unification

Constraint-based typing

Definition

- A **constraint set** C is a set of equations $\{ S_i = T_i^{i \in 1 \ldots n} \}$.
- A substitution σ is said to unify an equation $S = T$ if the substitution instances σS and σT are **identical**.
- We say that σ unifies C if it unifies **every equation** in C.

Definition

Suppose that $\Gamma \models t : S \mid_\chi C$. A solution for (Γ, t, S, C) is a pair (σ, T) s.t. σ satisfies C and $\sigma S = T$.

We read $\Gamma \models t : T \mid_\chi C$ as "Term t has type T under assumptions Γ whenever constraints C are satisfied"

LTH
FACULTY OF ENGINEERING

Constraint typing rules

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T \mid_{\varnothing} \{\}} \quad \text{(CT-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2 \mid_X C}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2 \mid_X C} \quad \text{(CT-ABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid_{X_1} C_1 \qquad \Gamma \vdash t_2 : T_2 \mid_{X_2} C_2 \\ X_1 \cap X_2 = X_1 \cap FV(T_2) = X_2 \cap FV(T_1) = \varnothing \\ X \notin X_1, X_2, T_1, T_2, C_1, C_2, \Gamma, t_1, \text{or } t_2 \\ C' = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}\end{array}}{\Gamma \vdash t_1\ t_2 : X \mid_{X_1 \cup X_2 \cup \{X\}} C'} \quad \text{(CT-APP)}$$

$$\Gamma \vdash 0 : \text{Nat} \mid_{\varnothing} \{\} \quad \text{(CT-ZERO)}$$

$$\frac{\Gamma \vdash t_1 : T \mid_X C \qquad C' = C \cup \{T = \text{Nat}\}}{\Gamma \vdash \text{succ } t_1 : \text{Nat} \mid_X C'} \quad \text{(CT-SUCC)}$$

$$\frac{\Gamma \vdash t_1 : T \mid_X C \qquad C' = C \cup \{T = \text{Nat}\}}{\Gamma \vdash \text{pred } t_1 : \text{Nat} \mid_X C'} \quad \text{(CT-PRED)}$$

$$\frac{\Gamma \vdash t_1 : T \mid_X C \qquad C' = C \cup \{T = \text{Nat}\}}{\Gamma \vdash \text{iszero } t_1 : \text{Bool} \mid_X C'} \quad \text{(CT-ISZERO)}$$

$$\Gamma \vdash \text{true} : \text{Bool} \mid_{\varnothing} \{\} \quad \text{(CT-TRUE)}$$

$$\Gamma \vdash \text{false} : \text{Bool} \mid_{\varnothing} \{\} \quad \text{(CT-FALSE)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid_{X_1} C_1 \\ \Gamma \vdash t_2 : T_2 \mid_{X_2} C_2 \qquad \Gamma \vdash t_3 : T_3 \mid_{X_3} C_3 \\ X_1, X_2, X_3 \text{ nonoverlapping} \\ C' = C_1 \cup C_2 \cup C_3 \cup \{T_1 = \text{Bool}, T_2 = T_3\}\end{array}}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \mid_{X_1 \cup X_2 \cup X_3} C'} \quad \text{(CT-IF)}$$

Declarative and Algorithmic specification

Given a context Γ and a term t, there are two ways of instantiating type variables in Γ and t to produce a valid typing:

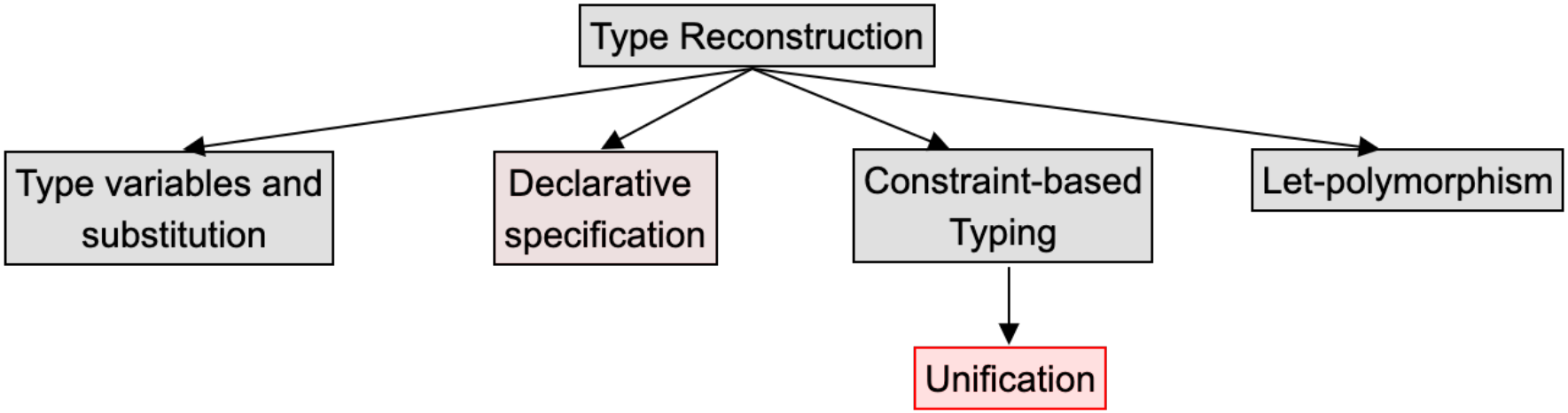
1) **[Declarative]**: as the set of all solutions for (Γ, t) in the sense of Definition 22.2.1
2) **[Algorithmic]**: via the constraint typing relation, by finding S and C such that $\Gamma \models t : S \mid C$ and then taking the set of solutions for (Γ, t, S, C).

The two specification are equivalent.

You can find the proof in the book :)

LTH
FACULTY OF ENGINEERING

Outline


Type Reconstruction
Type variables and substitution
Declarative specification
Constraint-based Typing
Let-polymorphism
Unification

Unification

To compute solutions to constraint sets, we use the idea of unification

$$
\begin{array}{lll}
\mathit{unify}(C) & = & \text{if } C = \emptyset, \text{ then } [\,] \\
& & \text{else let } \{\mathsf{S} = \mathsf{T}\} \cup C' = C \text{ in} \\
& & \quad \text{if } \mathsf{S} = \mathsf{T} \\
& & \quad\quad \text{then } \mathit{unify}(C') \\
& & \quad \text{else if } \mathsf{S} = \mathsf{X} \text{ and } \mathsf{X} \notin \mathit{FV}(\mathsf{T}) \\
& & \quad\quad \text{then } \mathit{unify}([\mathsf{X} \mapsto \mathsf{T}]C') \circ [\mathsf{X} \mapsto \mathsf{T}] \\
& & \quad \text{else if } \mathsf{T} = \mathsf{X} \text{ and } \mathsf{X} \notin \mathit{FV}(\mathsf{S}) \\
& & \quad\quad \text{then } \mathit{unify}([\mathsf{X} \mapsto \mathsf{S}]C') \circ [\mathsf{X} \mapsto \mathsf{S}] \\
& & \quad \text{else if } \mathsf{S} = \mathsf{S}_1 \to \mathsf{S}_2 \text{ and } \mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2 \\
& & \quad\quad \text{then } \mathit{unify}(C' \cup \{\mathsf{S}_1 = \mathsf{T}_1, \mathsf{S}_2 = \mathsf{T}_2\}) \\
& & \quad \text{else} \\
& & \quad\quad \mathit{fail}
\end{array}
$$

More about unification

- We say that a substitution σ is **less specific** than a substitution σ', written $\sigma \sqsubseteq \sigma'$, if $\sigma' = \gamma \circ \sigma$ for some substitution γ.
- σ is a **principal unifier** for a constraint set C iff

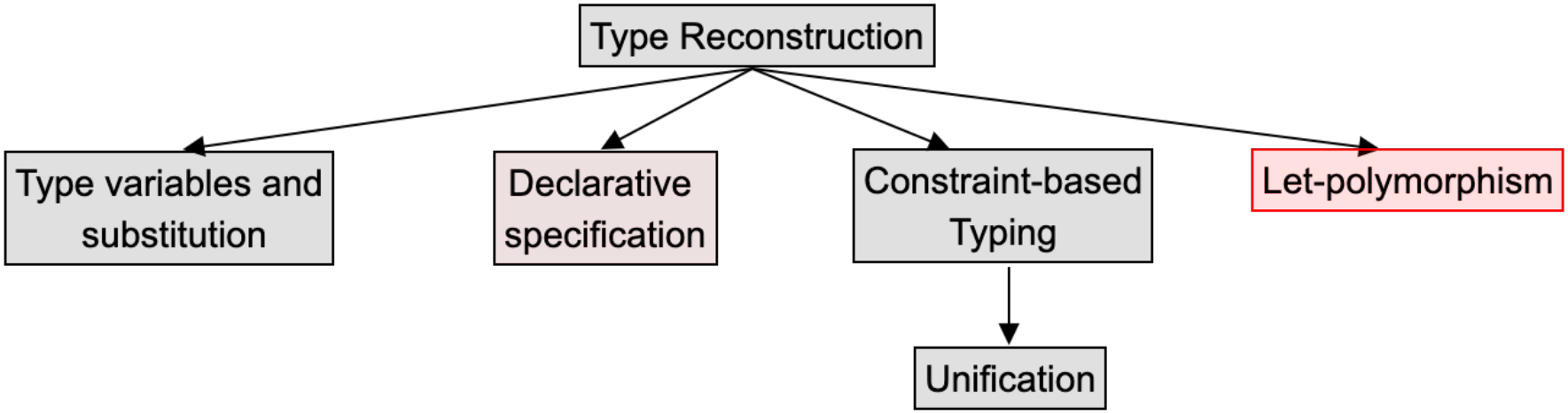
$$\forall \sigma' \in C : \sigma \sqsubseteq \sigma'.$$

Theorem

*The algorithm '**unify**' always terminates, failing when given a non unifiable constraint set as input and otherwise returning a principal unifier.*

LTH
FACULTY OF ENGINEERING

Outline


Type Reconstruction
Type variables and substitution
Declarative specification
Constraint-based Typing
Let-polymorphism
Unification

Let-polymorphism

The term polymorphism refers to a range of language mechanisms that allow a single part of a program to be used with different types.

Example

```
let f = \x.x in
let nat = f 1 in
let bool = f true in ...
```

- With the type reconstruction algorithms discussed previously, this program is not well-typed.
- The type reconstruction algorithm shown before can be generalized to provide a simple form of polymorphism known as let-polymorphism.
- More about polymorphism will be discussed next week.

LTH
FACULTY OF ENGINEERING

Let-polymorphism

Old typing rule:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2}$$

New typing rule:

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2} \qquad \frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \ \ |_X\ C}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2 \ \ |_X\ C}$$

Let-polymorphism

Old typing rule:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2}$$

New typing rule:

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2} \qquad \frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \ \ |_x\ C}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2 \ \ |_x\ C}$$

What about '`let x = <garbage> in 5`' ?

LTH
FACULTY OF ENGINEERING

Let-polymorphism

Old typing rule:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2}$$

New typing rule:

$$\frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2} \qquad \frac{\Gamma \vdash [x \mapsto t_1]t_2 : T_2 \ \ |_X\ C}{\Gamma \vdash \texttt{let}\ x{=}t_1\ \texttt{in}\ t_2 : T_2 \ \ |_X\ C}$$

What about '`let x = <garbage> in 5`' ?

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

LTH FACULTY OF ENGINEERING

Essentially linear

The algorithm is efficient ad in practice it appears "**essentially linear**" in the size of the input program. But the worst-case complexity is still exponential.

```
let f0 = fun x → (x,x) in
  let f1 = fun y → f0(f0 y) in
    let f2 = fun y → f1(f1 y) in
      let f3 = fun y → f2(f2 y) in
        let f4 = fun y → f3(f3 y) in
          let f5 = fun y → f4(f4 y) in
            f5 (fun z → z)
```

LTH
FACULTY OF ENGINEERING

Conclusion

Type reconstruction algorithm

=

Constraint generation

+

unification.

First introduction to **polymorphism**