Simple Type Inference

Announcements

- Quiz 5
- No class on April 8th
- I have graded HW4
- HW6 is a team homework
- I will work on paper list, guidelines and presentation schedule over weekend

So far

- Introduction to types and type systems
- Simply typed lambda calculus (System F_1)
  - Language syntax, type expression syntax
  - Static semantics
  - Dynamic semantics
  - Type soundness: Safety = Progress + Preservation
    - Proved for the simply typed lambda calculus

Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism (next time…)
- Hindley Milner type inference. Algorithm W

Reading

- "Types and Programming Languages", by Benjamin Pierce, Chapter 22, 23
- Lecture notes based partially on MIT 2015 Program Analysis OCW

Core Typing Rules

$$\frac{}{\Gamma \vdash c : \text{int}}$$

[x : int, y : int->int]

Type expressions:
$\tau ::= \text{int} \mid \tau \rightarrow \tau$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \quad (\text{Var})$$

Environment:
$\Gamma ::= \text{Nil} \mid \Gamma, x{:}\tau$

$\Gamma + x:\sigma$

$$\frac{\Gamma, x{:}\sigma \vdash E_1 : \tau}{\Gamma \vdash (\lambda x{:}\sigma.\ E_1) : \sigma \rightarrow \tau} \quad (\text{Abs})$$

$$\frac{\Gamma \vdash E_1 : \sigma \rightarrow \tau \quad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1\ E_2) : \tau} \quad (\text{App})$$

Extensions to Core Typing Rules

$$\frac{}{\Gamma \vdash c : int}$$

$$\frac{\Gamma \vdash E_1 : int \qquad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 + E_2 : int}$$

$$\frac{\Gamma \vdash E_1 : int \qquad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 = E_2 : bool}$$

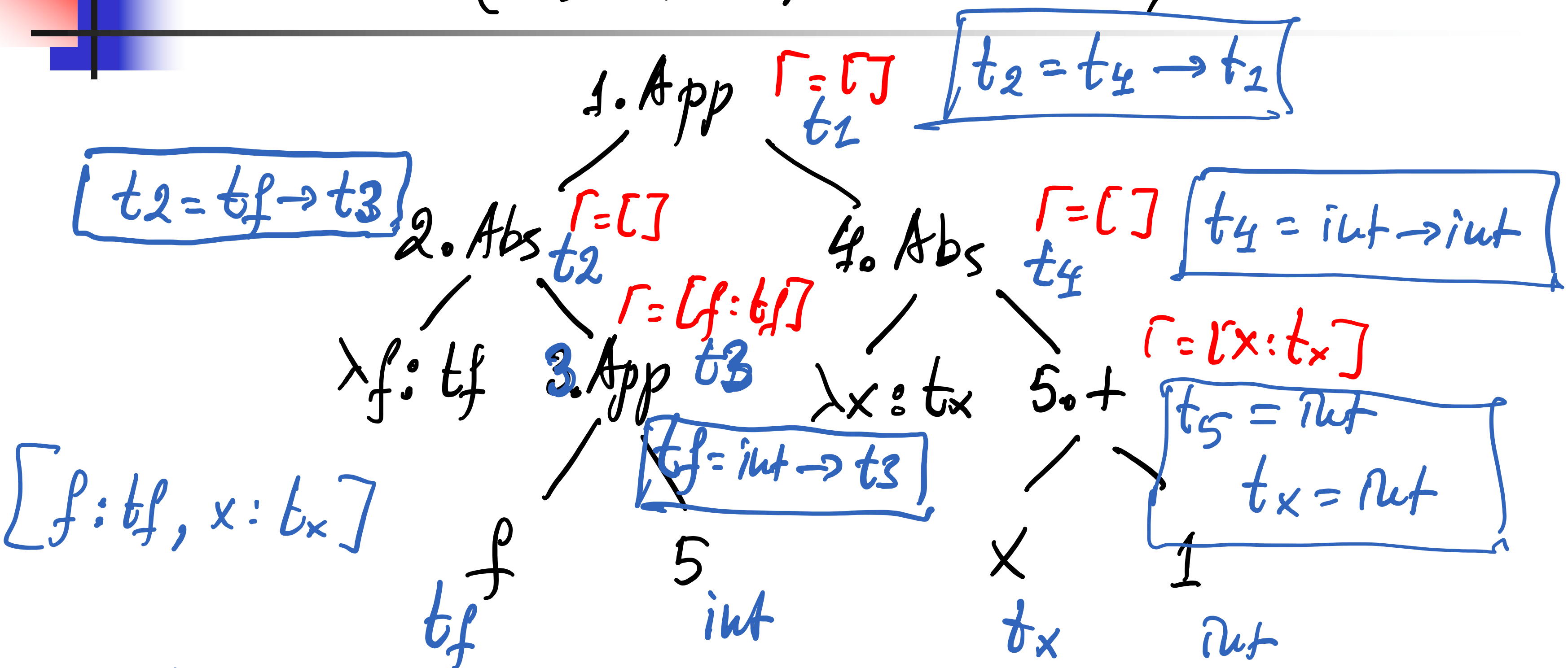
(Comparison)

$$\frac{\Gamma \vdash b : bool \qquad \Gamma \vdash E_1 : \tau \qquad \Gamma \vdash E_2 : \tau}{\Gamma \vdash \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau}$$

Type Inference, Strategy 1

- We can figure out all types even without explicit types for variables
  - **(λf. f 5) (λx. x+1) : ?**
  - Type inference
- Type inference, Strategy 1
  - Use typing rules to define type constraints
  - Solve type constraints
  - Aka constraint-based typing (e.g., Pierce)

$$Nil \vdash (\lambda f.\ f\ 5)(\lambda x.\ x+1) : t_1$$



$$t_2 = t_f \to t_3,\quad t_2 = t_4 \to t_1 \;\Rightarrow\; t_f = t_4,\quad t_3 = t_1$$

$$t_f = int \to t_3,\quad t_4 = int \to int \;\Rightarrow\; t_3 = int$$

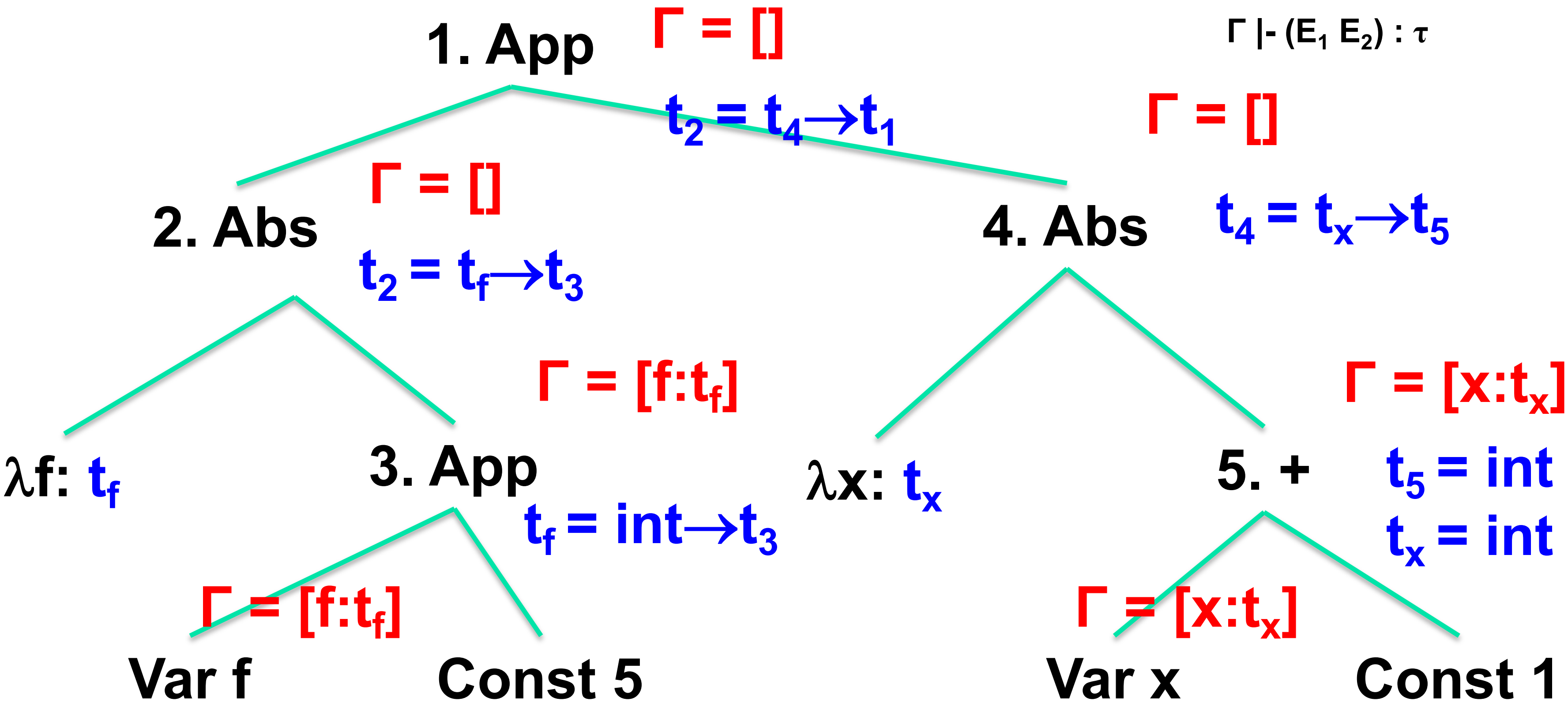
$$t_f = int \to int \;\Rightarrow\; t_1 = int$$

We Can Infer All Types!

- $(\lambda f.\ f\ 5)\ (\lambda x.\ x+1) : ?$

$$\frac{\Gamma \vdash E_1 : int \qquad \Gamma \vdash E_2 : int}{\Gamma \vdash E_1 + E_2 : int}$$

$$\frac{\Gamma \vdash E_1 : \sigma \to \tau \qquad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1\ E_2) : \tau}$$



Type Constraints

- We constructed a system of type constraints
- Let's solve the system of constraints

$t_2 = t_4 \rightarrow t_1$

$t_2 = t_f \rightarrow t_3$

$t_4 = t_x \rightarrow t_5$

$t_f = int \rightarrow t_3$

$t_5 = int,\ t_x = int$

$t_f = int \rightarrow t_3 = t_4 = int \rightarrow int$

$t_3 = int \quad t_1 = t_3 = int$

$t_4 = int \rightarrow int$

We inferred all **t**'s!

$t_1 = int$

$t_2 = (int \rightarrow int) \rightarrow int$

$t_3 = int$

$t_4 = int \rightarrow int$

$t_f = int \rightarrow int$

- **(λf:int→int. f 5) (λx:int. x+1) : int (t_1)**

$twice = \lambda f. \lambda x.\ f\ (f\ x)$

Another Example

$(\tau \to \tau) \to (\tau \to \tau)$

- **twice f x = f (f x)**
- What is the type of **twice**?

$t_f = t_x \to t_2 \qquad t_x \to t_x = t_f$

$t_f = t_2 \to t_1 \qquad t_x \to t_x = t_f$

$twice : (t_x \to t_x) \to t_x \to t_x$

Another Example

- **twice f x = f (f x)**
- What is the type of **twice**?
  - It is $\mathbf{t_f \to t_x \to t_1}$ ($\mathbf{t_1}$ is the type of **f (f x)**)
- Based on the syntax tree of **f (f x)** we have:

$\mathbf{t_f = t_2 \to t_1}$

$\mathbf{t_f = t_x \to t_2}$

Thus, $\mathbf{t_x = t_1 = t_2}$, $\mathbf{t_f = t_x \to t_x}$ and

type of **twice** is $\mathbf{(t_x \to t_x) \to t_x \to t_x}$

Note: $\mathbf{t_x}$ is a free type variable! Polymorphism!

Type Constraints from Typing Rules, as Attribute Grammar

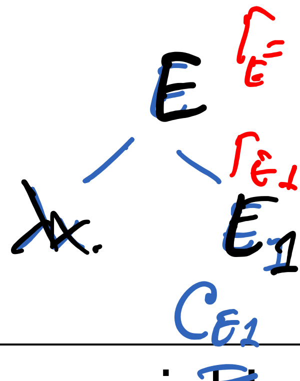
- Syntax: $\mathbf{E ::= x \mid c \mid \lambda x.E \mid E_1\ E_2 \mid E_1 + E_2}$

Grammar rule:	Attribute rule:
$\mathbf{E ::= x}$	$C_E = \{ t_E = \Gamma_E(\mathbf{x}) \}$
$\mathbf{E ::= c}$	$C_E = \{ t_E = \mathbf{int} \}$
$\mathbf{E ::= \lambda x.E_1}$	$\Gamma_{E1} = \Gamma_E;x:t_x$
	$C_E = C_{E1} \cup \{ t_E = t_x \rightarrow t_{E1} \}$
$\mathbf{E ::= E_1\ E_2}$	$\Gamma_{E1} = \Gamma_E \quad \Gamma_{E2} = \Gamma_E$
	$C_E = C_{E1} \cup C_{E2} \cup \{ t_{E1} = t_{E2} \rightarrow t_E \}$
$\mathbf{E ::= E_1 + E_2}$	$\Gamma_{E1} = \Gamma_E \quad \Gamma_{E2} = \Gamma_E$

$C_E = C_{E1} \cup C_{E2} \cup \{ t_{E1} = \mathbf{int},\ t_{E2} = \mathbf{int},\ t_E = \mathbf{int} \}$

Type Constraints from Typing Rules, as Attribute Grammar

…



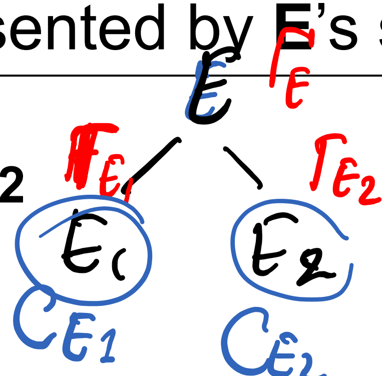
$E ::= \lambda x.E_1$

$\Gamma_{E1} = \Gamma_E; x:t_x$

$C_E = C_{E1} \cup \{\ t_E = t_x \rightarrow t_{E1}\ \}$

Γ is inherited. Propagates top-down the tree.

t_E is "fresh" type variable for term represented by E's subtree.



$E ::= E_1\ E_2$

$\Gamma_{E1} = \Gamma_E \quad \Gamma_{E2} = \Gamma_E$

$C_E = C_{E1} \cup C_{E2} \cup \{\ t_{E1} = t_{E2} \rightarrow t_E\ \}$…

C collects constraints. It is synthesized. Propagates bottom-up the tree.

Solving Constraints

- Two key concepts
- Equality
  - What does it mean for two types to be equal?
  - Structural equality (aka structural equivalence)
- Unification
  - Can two types be made equal by choosing appropriate substitutions for their type variables?
  - Robinson’s unification algorithm (which you already know from Prolog!)

Equality and Unification

- What does it mean for two types τ_a and τ_b to be equal?

- Structural equality
  - Suppose $\tau_a = t_1 \rightarrow t_2$
    $\tau_b = t_3 \rightarrow t_4$
  - Structural equality entails

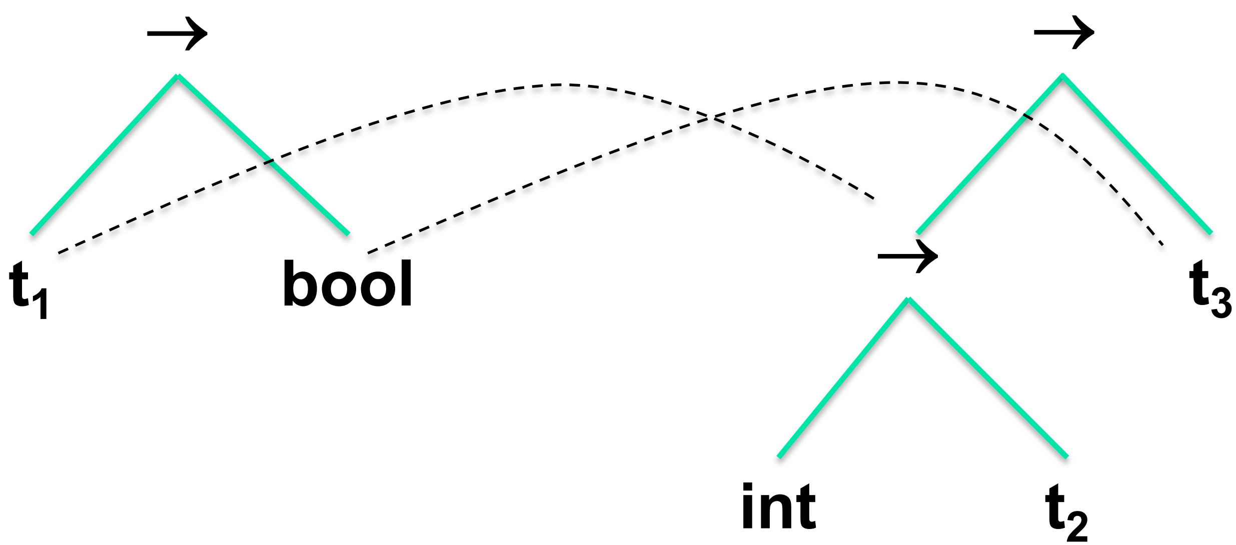
$\tau_a = \tau_b$ means $t_1 \rightarrow t_2 = t_3 \rightarrow t_4$ iff $t_1 = t_3$ and $t_2 = t_4$

Equality and Unification

- Can two types be made equal by choosing appropriate substitutions for their type variables?
- Robinson's unification algorithm
  - Suppose $\tau_a = \textbf{int} \rightarrow \textbf{t}_1$
    $\tau_b = \textbf{t}_2 \rightarrow \textbf{bool}$
  - Can we unify τ_a and τ_b? Yes, if **bool**/$\textbf{t}_1$ and **int**/$\textbf{t}_2$
  - Suppose $\tau_a = \textbf{int} \rightarrow \textbf{t}_1$
    $\tau_b = \textbf{bool} \rightarrow \textbf{bool}$
  - Can we unify τ_a and τ_b? No.

Example

$$t_1 \rightarrow \textbf{bool} \quad = \quad (\textbf{int} \rightarrow t_2) \rightarrow t_3$$



Yes, if $\textbf{int} \rightarrow t_2 / t_1$ and $\textbf{bool} / t_3$

Simple Type Substitution (essential to define unification)

- Language of types

  τ ::= **b** // primitive type, e.g., **int**, **bool**

  | **t** // type variable

  | $\tau \to \tau$ // function type

- A substitution is a map
  - **S : Type Variable → Type**
  - **S = [τ_1/t_1, … τ_n/t_n]** // substitute type τ_i for type var t_i
- A substitution instance **τ' = S τ**
  - **S = [$t_0 \to$bool / t_1]** **τ = $t_1 \to t_1$** then
  - **S(τ) = S($t_1 \to t_1$) = ($t_0 \to$bool) → ($t_0 \to$bool)**

Simple Type Substitution (essential to define unification)

- Substitutions can be composed
  - **S_1 = [$t_0 \rightarrow$bool / t_1]**
  - **S_2 = [int / t_0]**
  - **τ = $t_1 \rightarrow t_1$**
  - **S_2 S_1 (τ) = S_2 (S_1 ($t_1 \rightarrow t_1$)) =**

Examples

- Substitutions can be composed
  - $S_1 = [\ t_x\ /\ t_1\ ]$
  - $S_2 = [\ t_x\ /\ t_2\ ]$

  - $\tau = t_2 \rightarrow t_1$
  - $S_2\ S_1\ (\tau) = ?$

Examples

- Substitutions can be composed
  - $S_1 = [\ t_1 / t_2\ ]$
  - $S_2 = [\ t_3 / t_1\ ]$
  - $S_3 = [\ t_4 \rightarrow int / t_3\ ]$

  - $\tau = t_1 \rightarrow t_2$
  - $S_3\ S_2\ S_1\ (\tau) = ?$

Some Terminology...

- A substitution $\mathbf{S_1}$ is less specific (i.e., more general) than substitution $\mathbf{S_2}$ if $\mathbf{S_2 = S\ S_1}$ for some substitution $\mathbf{S}$
  - E.g., $\mathbf{S_1 = [\ t_1 \to t_1 \ /\ t_2\ ]}$ is more general than $\mathbf{S_2 = [\ int \to int\ /\ t_2\ ]}$ because $\mathbf{S_2 = S\ S_1}$ for $\mathbf{S = [\ int\ /\ t_1\ ]}$
- A principal unifier of a constraint set $\mathbf{C}$ is a substitution $\mathbf{S_1}$ that satisfies $\mathbf{C}$, and $\mathbf{S_1}$ is more general than any $\mathbf{S_2}$ that satisfies $\mathbf{C}$

Examples

- Find principal unifiers (when they exist) for
  - { $int \to int = t_1 \to t_2$ }
  - { $int = int \to t_2$ }
  - { $t_1 = int \to t_2$ }

  - { $t_1 = int$, $t_2 = t_1 \to t_1$ }
  - { $t_1 \to t_2 = t_2 \to t_3$, $t_3 = t_4 \to t_5$ }

Unification (essential for type inference!)

- **Unify**: tries to unify τ_1 and τ_2 and returns a **principal unifier for** $\tau_1 = \tau_2$ if unification is successful

This is the occurs check!

def **Unify**(τ_1,τ_2) =
  case (τ_1,τ_2)
    (τ_1,t_2) = [τ_1/t_2] provided t_2 does not occur in τ_1
    (t_1,τ_2) = [τ_2/t_1] provided t_1 does not occur in τ_2
    (b_1,b_2) = if (eq? b_1 b_2) then [ ] else ***fail***
    ($\tau_{11} \rightarrow \tau_{12}$, $\tau_{21} \rightarrow \tau_{22}$) = let S_1 = **Unify**(τ_{11},τ_{21})
                      S_2 = **Unify**($S_1(\tau_{12})$,$S_1(\tau_{22})$)
                  in S_2 S_1 // compose substitutions
    otherwise = ***fail***

Examples

- **Unify** (**int→int**, **t_1→t_2**) yields ?

- **Unify** (**int**, **int→t_2**) yields ?

- **Unify** (**t_1**, **int→t_2**) yields ?

Unify Set of Constraints **C**

- **UnifySet**: tries to unify **C** and returns a **principal unifier for C** if unification is successful

def **UnifySet** (**C**) =

if **C** is Empty Set then **[]**

else let

C = { $\tau_1=\tau_2$ } U **C'**

S = **Unify** (τ_1,τ_2) // **Unify** returns a substitution **S**

in

UnifySet (**S**(**C'**)) **S**

// Compose the substitutions

Examples

- $\{ t_1 = int, t_2 = t_1 \to t_1 \}$
- $\{ t_1 \to t_2 = t_2 \to t_3, t_3 = t_4 \to t_5 \}$
- $\{ t_f = t_2 \to t_1, t_f = t_x \to t_2 \}$
- $\{ t_2 = t_4 \to t_1, t_2 = t_f \to t_3, t_4 = t_x \to t_5, t_f = int \to t_3, t_5 = int, t_x = int \}$

Type Inference, Strategy 1

- Aka constraint-based typing (e.g., Pierce)
- Traverse parse tree to derive a set of type constraints **C**
  - These are equality constraints
  - (Pseudo code in earlier slides)
- Solve type constraints offline
  - Use unification algorithm
  - (Pseudo code in earlier slide)

Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism (next time…)
- Hindley Milner type inference. Algorithm W

Type Inference, Strategy 2

- Strategy 1 collects all constraints, then solves them offline
- Strategy 2 solves constraints on the fly
  - Builds the substitution map incrementally

Add a New Attribute, Substitution Map **S**

T_E is the inferred type of **E**.
S_E is the substitution map resulting from inferring T_E.
t_x, t_E are fresh type variables.

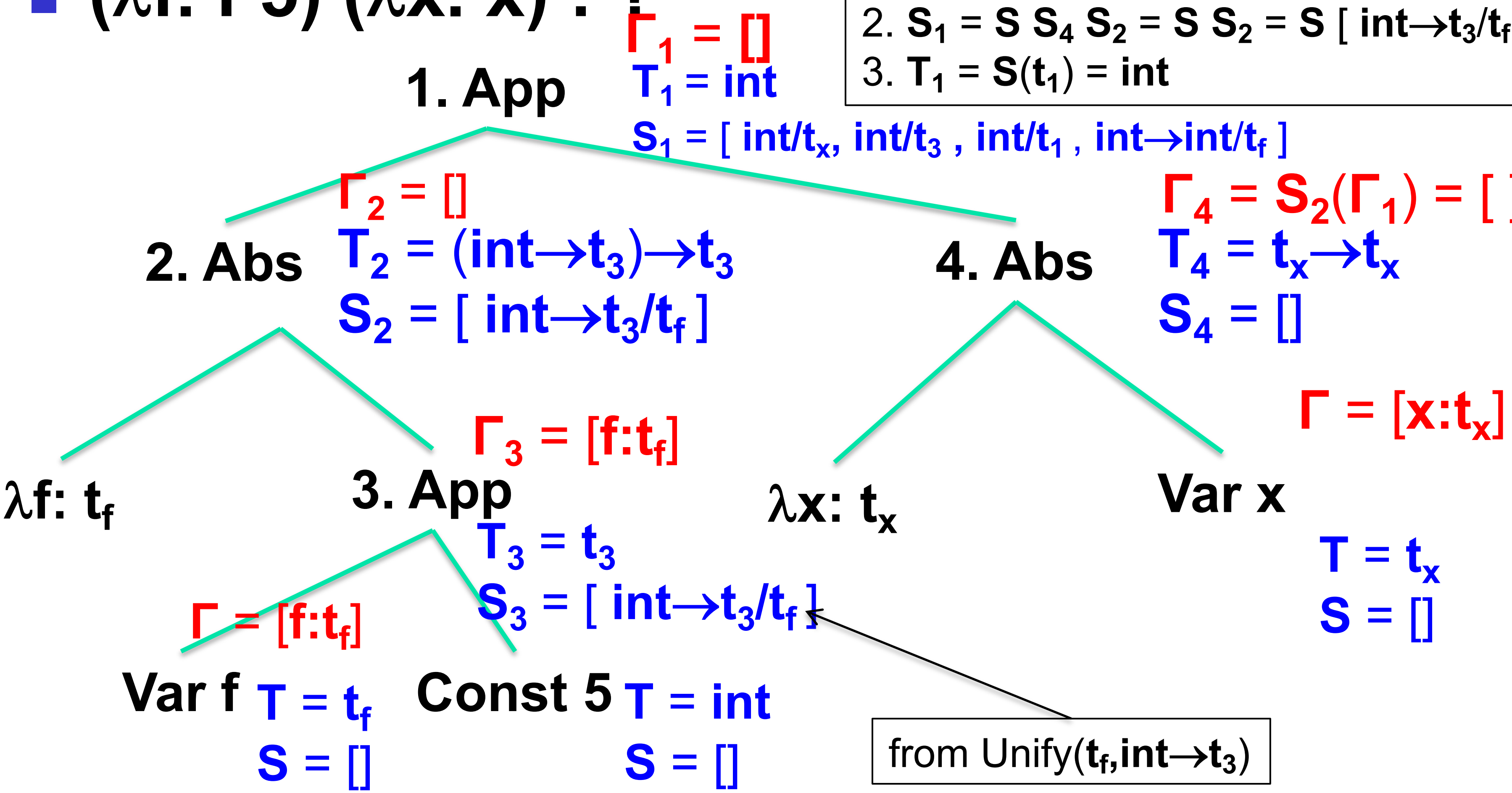
Grammar rule:	Attribute rule:
$E ::= x$	$T_E = \Gamma_E(x) \quad S_E = [\,]$
$E ::= c$	$T_E = int \quad S_E = [\,]$
$E ::= \lambda x.E_1$	$\Gamma_{E1} = \Gamma_E; x{:}t_x$ $T_E = S_{E1}(t_x) \to T_{E1} \quad S_E = S_{E1}$
$E ::= E_1\, E_2$	$\Gamma_{E1} = \Gamma_E \quad \Gamma_{E2} = S_{E1}(\Gamma_E)$ $S = Unify(S_{E2}(T_{E1}), T_{E2} \to t_E)$ $T_E = S(t_E) \quad S_E = S\; S_{E2}\; S_{E1}$

Example: (λf. f 5) (λx. x)

Steps at 1, finally:

1. unify($(int \to t_3) \to t_3$, $(t_x \to t_x) \to t_1$) returns S = [int/t_x, int/t_3, int/t_1]
2. $S_1 = S\ S_4\ S_2 = S\ S_2 = S$ [$int \to t_3/t_f$]
3. $T_1 = S(t_1) = int$

- (λf. f 5) (λx. x) : ?



Example: **λf.λx. (f (f x))**

The Let Construct

- In dynamic semantics, **let x = E_1 in E_2** is equivalent to **(λx.E_2) E_1**
- Typing rule

$$\frac{\Gamma \mid\!- E_1 : \sigma \qquad \Gamma;x{:}\sigma \mid\!- E_2 : \tau}{\Gamma \mid\!- \textbf{let } x = E_1 \textbf{ in } E_2 : \tau}$$

- In static semantics **let x = E_1 in E_2** is not equivalent to **(λx.E_2) E_1**
  - In **let**, the type of “argument” **E_1** is inferred/checked **before** the type of function body **E_2**
  - **let** construct enables Hindley Milner style polymorphism!

The Let Construct

- Typing rule

$$\frac{\Gamma \;|\text{-}\; E_1 : \sigma \qquad \Gamma; x{:}\sigma \;|\text{-}\; E_2 : \tau}{\Gamma \;|\text{-}\; \text{let } x = E_1 \text{ in } E_2 : \tau}$$

- Attribute grammar rule

$E ::= \text{let } x = E_1 \text{ in } E_2$

$\Gamma_{E1} = \Gamma_E$

$\Gamma_{E2} = S_{E1}(\Gamma_E) + \{x{:}T_{E1}\}$

$T_E = T_{E2}$ $\quad S_E = S_{E2}\, S_{E1}$

The Letrec Construct

- **letrec x = E_1 in E_2**
  - **x** can be referenced from within **E_1**
  - Extends calculus with general recursion
    - No need to type **fix** (we can't!) but we can still type recursive functions like **plus**, **times**, etc.
  - Haskell's **let** is a **letrec** actually…
- E.g.,

letrec plus = λx.λy. if (x=0) then y else ((plus x-1) y+1)

written as

letrec plus x y = if (x=0) then y else plus (x-1) (y+1)

The Letrec Construct

- **letrec x = E_1 in E_2**

Extensions over let rule
1. T_{E1} is inferred in augmented environment $\Gamma_E + \{x:t_x\}$
2. Must unify $S_{E1}(t_x)$ and T_{E1}
3. Apply substitution **S** on top of S_{E1}

Note: Can merge **let** and **letrec**, in **let** **Unify** and **S** have no impact

- Attribute grammar rule

E ::= letrec x = E_1 in E_2

$$\Gamma_{E1} = \Gamma_E + \{x:t_x\}$$

$$S = \mathbf{Unify}(S_{E1}(t_x), T_{E1})$$

$$\Gamma_{E2} = S\ S_{E1}(\Gamma_E) + \{x:T_{E1}\}$$

$$T_E = T_{E2} \qquad S_E = S_{E2}\ S\ S_{E1}$$

let/letrec Examples

letrec plus x y = if (x=0) then y else plus (x-1) (y+1)

- Typing **plus** using Strategy 1…

  $\mathbf{t_{plus}} = \mathbf{t_x \rightarrow t_y \rightarrow t_1}$

  $\mathbf{t_x}$ = **int** // because of **x=0** and **x-1**

  $\mathbf{t_y}$ = **int** // because of **y+1**

  Unify($\mathbf{t_{plus}}$, **int→int→int**) yields $\mathbf{t_1}$ = **int**

- Haskell

  plus :: int -> int -> int

  plus x y = if (x=0) then y else plus (x-1) (y+1)

Algorithm W, Almost There!

def W(Γ, E) = case E of

$c \;\rightarrow\; ([\,], \text{TypeOf}(c))$

$x \;\rightarrow\;$ if (x NOT in Dom(Γ)) then *fail*

else let $T_E = \Gamma(x)$;

in $([\,], T_E)$

$\lambda x.E_1 \;\rightarrow$ let $(S_{E1},T_{E1}) = W(\Gamma+\{x:t_x\},E_1)$

in $(S_{E1}, S_{E1}(t_x)\rightarrow T_{E1})$

$E_1\ E_2 \;\rightarrow$ let $(S_{E1},T_{E1}) = W(\Gamma,E_1)$

$(S_{E2},T_{E2}) = W(S_{E1}(\Gamma),E_2)$

$S = \text{Unify}(S_{E2}(T_{E1}),T_{E2}\rightarrow t)$

in $(S\ S_{E2}\ S_{E1}, S(t))$ // $S\ S_{E2}\ S_{E1}$ composes substitutions

let $x = E_1$ in $E_2 \;\rightarrow$ let $(S_{E1},T_{E1}) = W(\Gamma,E_1)$

$(S_{E2},T_{E2}) = W(S_{E1}(\Gamma)+\{x:T_{E1}\},E_2)$

in $(S_{E2}\ S_{E1}, T_{E2})$

Algorithm W, Almost There! (merges let and letrec)

def W(Γ, E) = case E of

c -> ([], TypeOf(c))

x -> if (x NOT in Dom(Γ)) then *fail*
else let $T_E = \Gamma(x)$;
in ([], T_E)

$\lambda x.E_1$ -> let $(S_{E1},T_{E1}) = W(\Gamma+\{x:t_x\},E_1)$
in $(S_{E1}, S_{E1}(t_x) \to T_{E1})$

$E_1\ E_2$ -> let $(S_{E1},T_{E1}) = W(\Gamma,E_1)$
$(S_{E2},T_{E2}) = W(S_{E1}(\Gamma),E_2)$
$S = \text{Unify}(S_{E2}(T_{E1}),T_{E2} \to t)$
in $(S\ S_{E2}\ S_{E1}, S(t))$ // $S\ S_{E2}\ S_{E1}$ composes substitutions

let x = E_1 in E_2 -> let $(S_{E1},T_{E1}) = W(\Gamma+\{x:t_x\},E_1)$
$S = \text{Unify}(S_{E1}(t_x),T_{E1})$
$(S_{E2},T_{E2}) = W(S\ S_{E1}(\Gamma)+\{x:T_{E1}\},E_2)$
in $(S_{E2}\ S\ S_{E1}, T_{E2})$

Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism
- Hindley Milner type inference. Algorithm W