Simple Type Inference, continued

Announcements

- CI designation approved
- Papers and guidelines coming up
- HW4, HW5, and Quiz 5 grades coming up

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 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)

Type Constraints

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

$t_2 = t_4 \rightarrow t_1$

$t_2 = t_f \rightarrow t_3$

$t_4 = t_x \rightarrow t_5$

$t_f = \text{int} \rightarrow t_3$

$t_5 = \text{int},\ t_x = \text{int}$

- $(\lambda f{:}\text{int} \rightarrow \text{int}.\ f\ 5)\ (\lambda x{:}\text{int}.\ x+1) : \text{int}\ (t_1)$

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 $\boldsymbol{\tau_a} = \mathbf{int} \to \mathbf{t_1}$

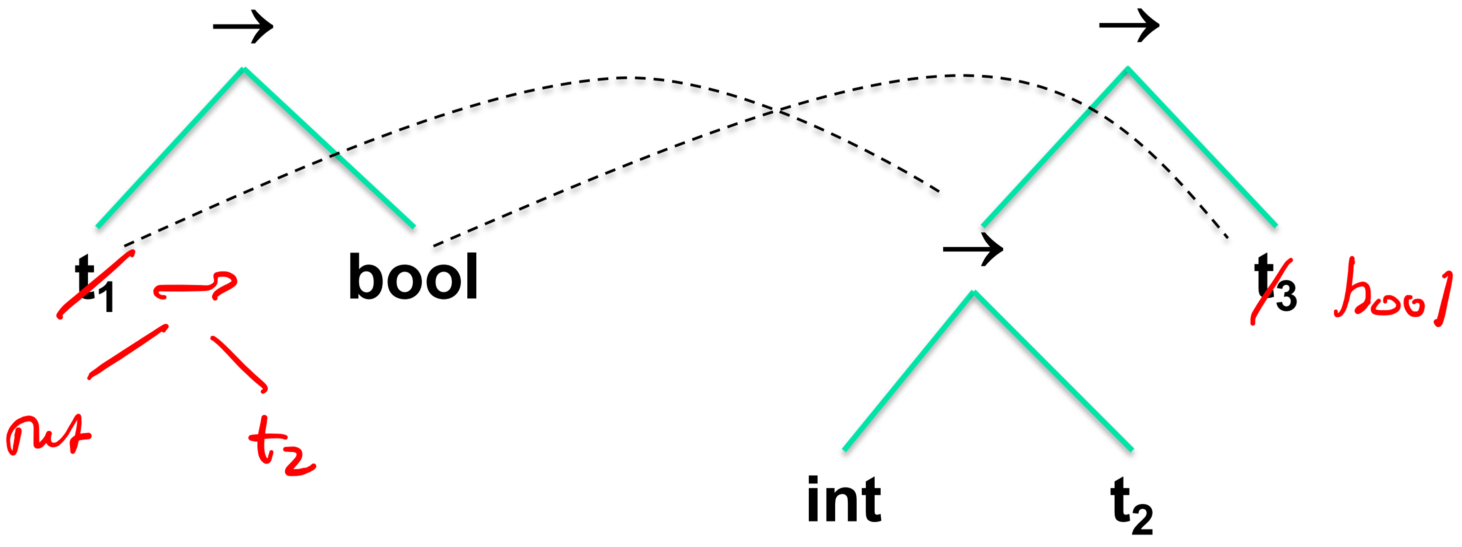
    $\boldsymbol{\tau_b} = \mathbf{t_2} \to \mathbf{bool}$
  - Can we unify $\boldsymbol{\tau_a}$ and $\boldsymbol{\tau_b}$? Yes, if $\mathbf{bool/t_1}$ and $\mathbf{int/t_2}$
  - Suppose $\boldsymbol{\tau_a} = \mathbf{int} \to \mathbf{t_1}$

    $\boldsymbol{\tau_b} = \mathbf{bool} \to \mathbf{bool}$
  - Can we unify $\boldsymbol{\tau_a}$ and $\boldsymbol{\tau_b}$? No.

Example

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

$\text{int} \rightarrow t_2 / t_1, \ \text{bool} / t_3$



Yes, if **int→t_2/t_1** and **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 $[\tau_n/t_n] \cdots \circ [\tau_2/t_2] \circ [\tau_1/t_1]$
  - **S : Type Variable → Type**
  - $\mathbf{S} = \underline{[\tau_1/t_1, \ldots \tau_n/t_n]}$ // substitute type τ_i for type var t_i
- A substitution instance $\tau' = \mathbf{S}\ \tau$
  - $\mathbf{S} = [\ t_0 \to \mathbf{bool}\ /\ t_1\ ]$ $\quad \tau = t_1 \to t_1$ then
  - $\mathbf{S}(\tau) = \mathbf{S}(t_1 \to t_1) = (t_0 \to \mathbf{bool}) \to (t_0 \to \mathbf{bool})$

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 = $t_1 \rightarrow t_1$**
  - **$S_2\ S_1(\tau) = S_2(\ S_1(t_1 \rightarrow t_1)\ )$ =** $(\text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow \text{bool}$

$S = [\ t_0 \rightarrow \text{bool} / t_1,\ \text{int} / t_0\ ]$

$S_1 = [\ t_0 \rightarrow \text{bool} / t_1,\ \text{int} \rightarrow \text{int} / \underline{t_4}\ ]$

Examples

- Substitutions can be composed
  - **$S_1 = [\, t_x / t_1 \,]$**
  - **$S_2 = [\, t_x / t_2 \,]$**
  - **$\tau = \overset{t_x}{\cancel{t_2}} \to \overset{t_x}{\cancel{t_1}}$**
  - **$S_2\, S_1\, (\tau) = ?$** $t_x \to t_x$

Examples

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

$S = [\, t_1 / t_2,\ t_3 / t_1,\ t_4 \rightarrow \text{int} / t_3 \,]$

  - **$\tau = \overset{t_4}{\cancel{t_1}} \rightarrow \overset{\overset{t_3}{\cancel{t_1}}}{\cancel{t_2}}$**
  - **$S_3\, S_2\, S_1\, (\tau) = ?$**

$(t_4 \rightarrow \text{int}) \rightarrow t_4 \rightarrow \text{int}$

$S_3(S_2(S_1(t_1 \rightarrow t_2))) =$

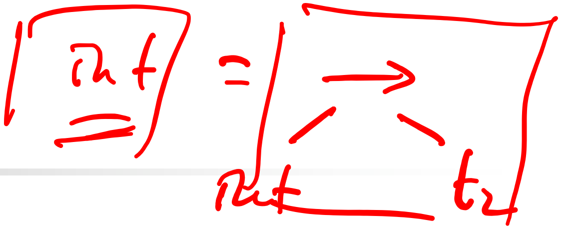
$S_3(S_2(t_1 \rightarrow t_1)) = S_3(t_3 \rightarrow t_3) = \cancel{\text{int}}\ (t_4 \rightarrow \text{int}) \rightarrow t_4 \rightarrow \text{int}$

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., $\mathbf{S_1 = [\ t_1 \rightarrow t_1 \ /\ t_2\ ]}$ is more general than $\mathbf{S_2 = [\ int \rightarrow int\ /\ t_2\ ]}$ because $\mathbf{S_2 = [\ int\ /\ t_1\ ]\ S_1}$

- 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
  - $\{ \text{int} \to \text{int} = t_1 \to t_2 \}$ [int/t_1, int/t_2]
  - $\{ \text{int} = \text{int} \to t_2 \}$ DOES NOT EXIST
  - $\{ t_1 = \text{int} \to t_2 \}$ [int→t_2 / t_1] → most general

    [int→t_2/t_1, bool→bool/t_2] → more specific
  - $\{ t_1 = \text{int}, t_2 = t_1 \to t_1 \}$ [int/t_1, int→int/t_2]
  - $\{ t_1 \to \overset{t_1}{\cancel{t_2}} = t_2 \to t_3, \overset{t_1}{\cancel{t_3}} = t_4 \to t_5 \}$

    [t_1/t_2, t_1/t_3, t_4→t_5/t_1]

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($\tau_1$,$\tau_2$) =
  case ($\tau_1$,$\tau_2$)
    ($\tau_1$,$t_2$) = [$\tau_1$/$t_2$] provided $t_2$ does not occur in $\tau_1$
    ($t_1$,$\tau_2$) = [$\tau_2$/$t_1$] provided $t_1$ does not occur in $\tau_2$
    ($b_1$,$b_2$) = if (eq? $b_1$ $b_2$) then [ ] else fail
    ($\tau_{11} \to \tau_{12}$, $\tau_{21} \to \tau_{22}$) = let  $S_1$ = Unify($\tau_{11}$,$\tau_{21}$)
                                   $S_2$ = Unify($S_1(\tau_{12})$,$S_1(\tau_{22})$)
                              in $S_2$ $S_1$ // compose substitutions
    otherwise = fail
```

but, $t_2 \to t_3$

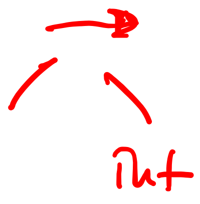
S_1 ++ S_2

Examples

$t_1 \to \overset{t_1}{\cancel{t_2}}, \; t_2 \to t_1$

$[t_1/t_2]$

$t_1 \overset{?}{=}$



- **Unify (int→int, t$_1$→t$_2$)** yields ?

$[int/t_1, int/t_2]$

- **Unify (int, int→t$_2$)** yields ?

fail

- **Unify (t$_1$, int→t$_2$)** yields ?

$[int \to t_2 / t_1]$

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 ($\tau_1,\tau_2$) // Unify returns a substitution S
      in
        UnifySet ( $\underline{S(C’)}$ ) S
        // Compose the substitutions
```

Examples

- $\{ t_1 = \text{int}, t_2 = \overset{\text{int}}{\cancel{t_1}} \rightarrow \overset{\text{int}}{\cancel{t_1}} \}$

$[\text{int}/t_1,\ \text{int} \rightarrow \text{int}/t_2]$

- $\{ t_1 \rightarrow \overset{t_1}{\cancel{t_2}} = t_2 \rightarrow t_3,\ \overset{t_1}{\cancel{t_3}} = t_4 \rightarrow t_5 \}$

$[t_1/t_2,\ t_1/t_3,\ t_4 \rightarrow t_5/t_1]$

$C = \{ \underline{t_f = t_2 \rightarrow t_1},\ \overset{t_2 \rightarrow t_1}{\cancel{t_f}} = t_x \rightarrow t_2 \}$

$S = [t_2 \rightarrow t_1/t_f,\ t_2/t_x,\ t_1/t_2]$

$S(C) = \{ \underline{t_1 \rightarrow t_1 = t_1 \rightarrow t_1},\ t_1 \rightarrow t_1 = t_1 \rightarrow t_1 \}$

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

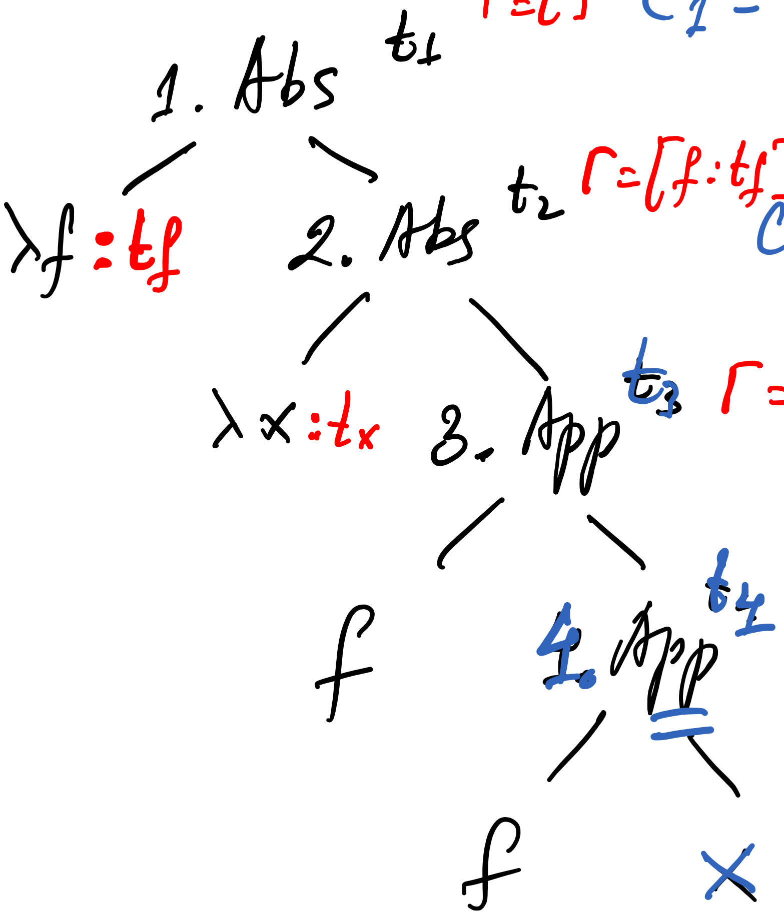
Type Inference, Strategy 1

- 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, Lecture 19)
- Solve type constraints offline
  - Use unification algorithm
  - (Pseudo code in earlier slide, this lecture)

Example: **λf.λx. (f (f x))**

- Creating constraints



$\Gamma = [\,]$ $C_1 = \{t_1 = t_f \rightarrow t_2\} \cup C_2$

$\Gamma = [f : t_f]$ $C_2 = \{t_2 = t_x \rightarrow t_3\} \cup C_3$

$\Gamma = [x : t_x, f : t_f]$ $C_3 = \{t_f = t_4 \rightarrow t_3\} \cup C_4$

$C_4 = \{t_f = t_x \rightarrow t_4\}$

Example: **λf.λx. (f (f x))**

$$t_x \to \overset{t_x\,?}{\cancel{t_4}} = t_4 \to t_3$$

$$[t_x/t_4,\ t_x/t_3]$$

- Solving constraints offline

$$C = \{ t_f = t_x \to t_4,\ \overset{t_x \to t_4}{\cancel{t_f}} = t_4 \to t_3,\ t_2 = t_x \to \overset{t_x}{\cancel{t_3}},\ t_1 = \overset{t_x \to t_x}{\cancel{t_f}} \to \underset{t_x \to t_x}{\cancel{t_2}} \}$$

$$S = [t_x \to t_4/t_f,\ t_x/t_4,\ t_x/t_3,\ t_x \to t_x/t_2,$$
$$(t_x \to t_x) \to t_x \to t_x / t_1]$$

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**

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

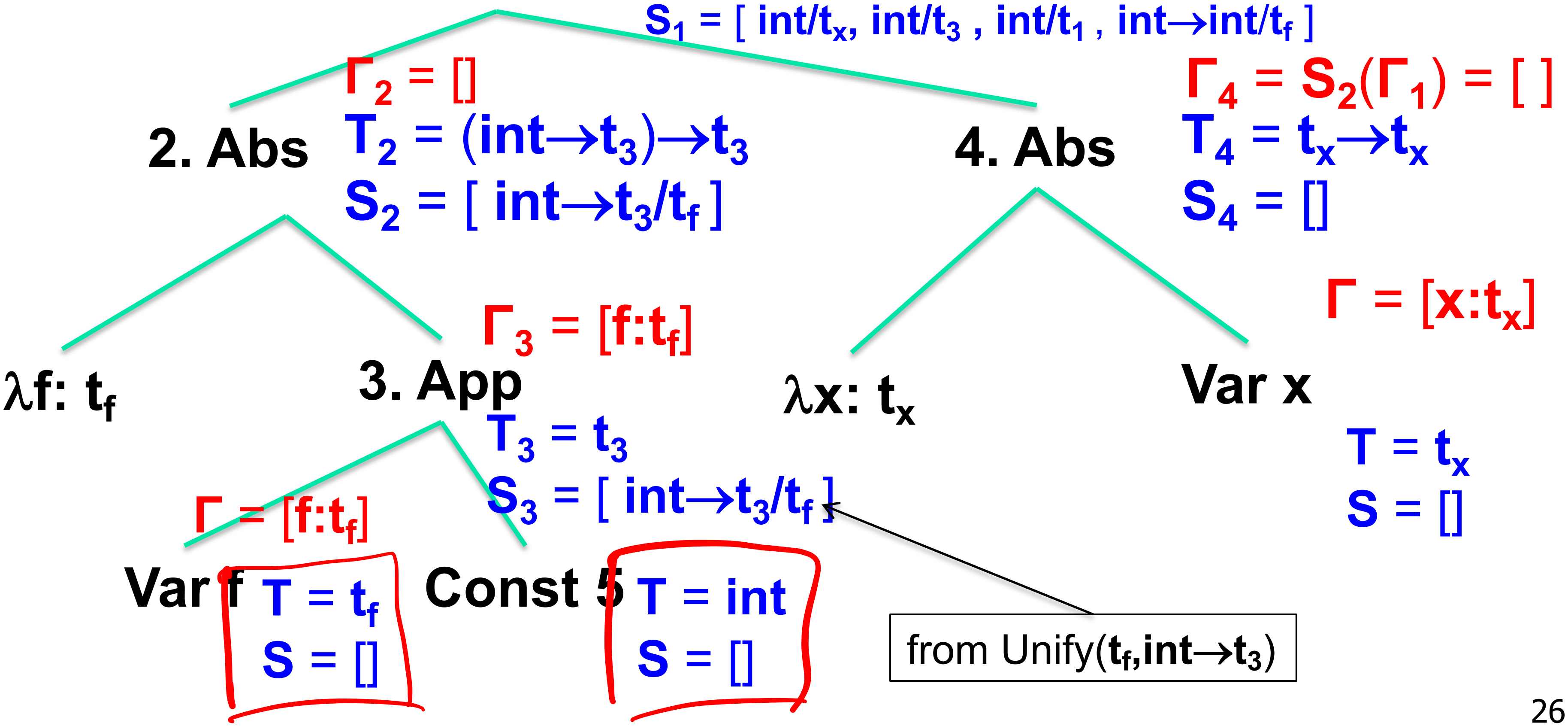
Grammar rule:	Attribute rule:
$\mathbf{E ::= x}$	$\mathbf{T_E = \Gamma_E(x)} \quad \mathbf{S_E = [\,]}$
$\mathbf{E ::= c}$	$\mathbf{T_E = int} \quad \mathbf{S_E = [\,]}$
$\mathbf{E ::= \lambda x.E_1}$	$\mathbf{\Gamma_{E1} = \Gamma_E;x:t_x}$ $\mathbf{T_E = S_{E1}(t_x)\rightarrow T_{E1}} \quad \mathbf{S_E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

Example: (λf. f 5) (λx. x)

- **(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. $T_1 = S(t_1) = int$



$\Gamma_1 = []$

1. App

$T_1 = int$

$S_1 = [\ int/t_x, int/t_3\ , int/t_1\ , int \to int/t_f\ ]$

$\Gamma_2 = []$

2. Abs

$T_2 = (int \to t_3) \to t_3$

$S_2 = [\ int \to t_3/t_f\ ]$

$\Gamma_4 = S_2(\Gamma_1) = [\ ]$

4. Abs

$T_4 = t_x \to t_x$

$S_4 = []$

λf: t_f

$\Gamma_3 = [f{:}t_f]$

3. App

$T_3 = t_3$

$S_3 = [\ int \to t_3/t_f\ ]$

λx: t_x

$\Gamma = [x{:}t_x]$

Var x

$T = t_x$

$S = []$

$\Gamma = [f{:}t_f]$

Var f

$T = t_f$

$S = []$

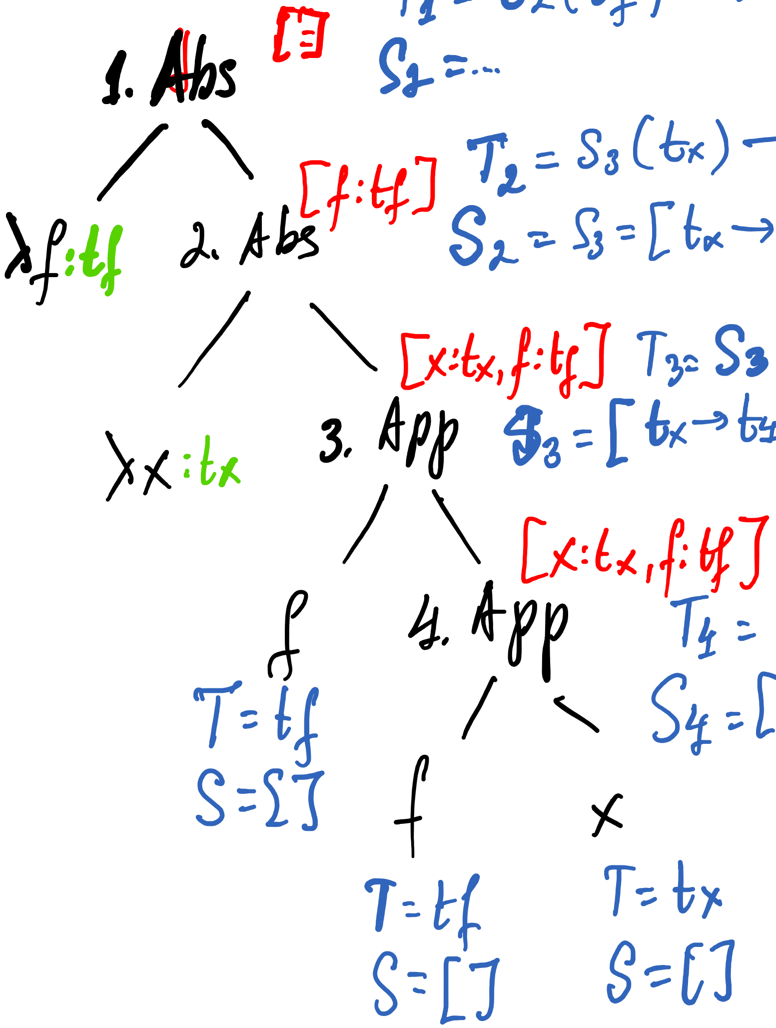
Const 5

$T = int$

$S = []$

from Unify($t_f, int \to t_3$)

Example: λf.λx. (f (f x))



$T_1 = S_2(t_f) \rightarrow t_x \rightarrow t_x \equiv \boxed{(t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x}$

$S_1 = \ldots$

(We solved for T_1 while traversing!)

$T_2 = S_3(t_x) \rightarrow t_x \equiv t_x \rightarrow t_x$

$S_2 = S_3 = [t_x \rightarrow t_4 / t_f, t_x / t_4, t_x / t_3]$

$T_3 = S_3(t_3) = t_x$

$S_3 = [t_x \rightarrow t_4 / t_f, t_x / t_4, t_x / t_3]$

$\rightarrow Unify(S_4(t_f), t_4 \rightarrow t_3)$

$T_4 = t_4$

$S_4 = [t_x \rightarrow t_4 / t_f]$

$\rightarrow Unify(t_f, t_x \rightarrow t_4)$

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

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

- Attribute grammar rule

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

$\Gamma_{E1} = \Gamma_E$

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

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

The Letrec Construct

- **letrec x = $\mathbf{E_1}$ in $\mathbf{E_2}$**
  - **x** can be referenced from within $\mathbf{E_1}$
  - 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. $\mathbf{T_{E1}}$ is inferred in augmented environment Γ_E + {**x**:t_x}
2. Must unify $S_{E1}(t_x)$ and $\mathbf{T_{E1}}$
3. Apply substitution **S** on top of $\mathbf{S_{E1}}$

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                  -> ([], 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 $(S_{E1}, S_{E1}(t_x) \rightarrow T_{E1})$
    $E_1\ E_2$         -> let $(S_{E1},T_{E1}) = W(\Gamma,E_1)$
                              $(S_{E2},T_{E2}) = W(S_{E1}(\Gamma),E_2)$
                              $S = Unify(S_{E2}(T_{E1}),T_{E2} \rightarrow t)$
                          in $(S\ S_{E2}\ S_{E1}, S(t))$ // $S\ S_{E2}\ S_{E1}$ composes substitutions
    let x = $E_1$ in $E_2$ -> 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 $(S_{E1},T_{E1}) = W(\Gamma,E_1)$
                          $(S_{E2},T_{E2}) = W(S_{E1}(\Gamma),E_2)$
                          $S = Unify(S_{E2}(T_{E1}),T_{E2}\to t)$
                      in $(S\ S_{E2}\ S_{E1}, S(t))$ // $S\ S_{E2}\ S_{E1}$ composes substitutions
    let x = $E_1$ in $E_2$ -> let $(S_{E1},T_{E1}) = W(\Gamma+\{x:t_x\},E_1)$
                              $S = Unify(S_{E1}(t_x),T_{E1})$
                              $(S_{E2},T_{E2}) = W(S\ S_{E1}(\Gamma)+\{x:T_{E1}\},E_2)$
                          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 (next time)
- Hindley Milner type inference. Algorithm W