Simple Type Inference, cont.
Announcements

- HW6 is a team homework
- Presentation guidelines are up, papers will be up on Schedule page by the end of week
  - 1. Select available paper/slot from list (first-come-first-serve)
  - 2. If available, I’ll assign and update, otherwise goto 1

- Quiz 5
Outline

- Simple type inference, cont.
  - Equality constraints, unification and substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing
  - The Let construct

- Examples in context of HW6 starter code

- Hindley Milner: next week
Unification
(essential for type inference!)

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

```python
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} \rightarrow \tau_{12}, \tau_{21} \rightarrow \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
```

This is the **occurs check**!
Examples

- \textbf{Unify} \(( \text{int} \rightarrow \text{int}, \, \mathit{t}_1 \rightarrow \mathit{t}_2 \) \) yields ?

- \textbf{Unify} \(( \, \text{int}, \, \text{int} \rightarrow \mathit{t}_2 \) \) yields ?

- \textbf{Unify} \(( \, \mathit{t}_1, \, \text{int} \rightarrow \mathit{t}_2 \) \) yields ?
Unify Set of Constraints $C$

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

```python
def UnifySet (C) =
    if C is Empty Set then []
    else let
        C = \{ \tau_1 = \tau_2 \} \cup C'
        S = Unify (\tau_1, \tau_2) // Unify returns a substitution $S$
in
        UnifySet ( S(C') ) \circ S
        // Composition of substitutions
```
Examples

- \{ t_1 = \text{int}, t_2 = t_1 \rightarrow t_1 \} \\

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

- \{ t_f = t_2 \rightarrow t_1, t_f = t_x \rightarrow t_2 \}
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 Lecture20)

- Solve type constraints offline
  - Use unification algorithm
  - (Pseudo code in Lecture20)
Outline

- Simple type inference, cont.
  - Equality constraints, Unification, Substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing
  - Let expressions

- Examples in context of HW6

- Hindley Milner: next week
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
  - Key reason: infers types as parser parses program!
Add a New Attribute, Substitution Map $S$

Grammar rule: \[
E ::= x
\]
\[
E ::= c
\]
\[
E ::= \lambda x.E_1
\]
\[
E ::= E_1 E_2
\]

Attribute rule: \[
T_E = \Gamma_E(x) \quad S_E = [ ]
\]
\[
T_E = \text{int} \quad S_E = [ ]
\]
\[
\Gamma_{E_1} = \Gamma_E; x:t_x
\]
\[
T_E = S_{E_1}(t_x) \rightarrow T_{E_1} \quad S_E = S_{E_1}
\]
\[
\Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = S_{E_1}(\Gamma_E)
\]
\[
S = \text{Unify}(S_{E_2}(T_{E_1}), T_{E_2} \rightarrow t_E)
\]
\[
T_E = S(t_E) \quad S_E = S \quad S_{E_2} \quad S_{E_1}
\]

$T_E$ 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.

Program Analysis CSCI 4450/6450, A Milanova
Example: $(\lambda f. f \, 5) \, (\lambda x. \, x)$

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

1. **App**
   - $\Gamma_1 = []$
   - $T_1 = \text{int}$
   - $S_1 = [\text{int} / t_x, \text{int} / t_3, \text{int} / t_1, \text{int} \rightarrow \text{int} / t_f]$

2. **Abs**
   - $\Gamma_2 = []$
   - $T_2 = (\text{int} \rightarrow t_3) \rightarrow t_3$
   - $S_2 = [\text{int} \rightarrow t_3 / t_f]$

3. **App**
   - $\Gamma_3 = [f : t_f]$
   - $T_3 = t_3$
   - $S_3 = [\text{int} \rightarrow t_3 / t_f]$

4. **Abs**
   - $\Gamma_4 = S_2(\Gamma_1) = []$
   - $T_4 = t_x \rightarrow t_x$
   - $S_4 = []$

- $\Gamma = [x : t_x]$

$T = t_x$
$S = []$

from Unify$(t_f, \text{int} \rightarrow t_3)$
The Let Construct

- In dynamic semantics, \texttt{let x = E\_1 in E\_2} is equivalent to \((\lambda x. E\_2) E\_1\)

- Typing rule

  \[
  \Gamma |- E\_1 : \sigma \quad \Gamma; x: \sigma |- E\_2 : \tau \\
  \hline
  \Gamma |- \texttt{let x = E\_1 in E\_2} : \tau
  \]

- In static semantics \texttt{let x = E\_1 in E\_2} is not equivalent to \((\lambda x. E\_2) E\_1\)
  - In \texttt{let}, the type of “argument” \(E\_1\) is inferred/checked \textbf{before} the type of function body \(E\_2\)
  - \texttt{let} construct enables Hindley Milner style polymorphism!
The Let Construct

- Typing rule

\[ \Gamma |- E_1 : \sigma \quad \Gamma; x:\sigma |- E_2 : \tau \]
\[ \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau \]

- Attribute grammar rule

\[ E ::= \text{let } x = E_1 \text{ in } E_2 \]
\[ \Gamma_{E_1} = \Gamma_E \]
\[ \Gamma_{E_2} = S_{E_1}(\Gamma_E) + \{ x : T_{E_1} \} \]
\[ T_E = T_{E_2} \quad S_E = S_{E_2} S_{E_1} \]
The Letrec Construct

- letrec x = E₁ in E₂
  - x can be referenced from within E₁
  - 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 = \x.\y. if (x=0) then y else ((plus x-1) y+1) in …

  or in Haskell syntax:

  let plus x y = if (x=0) then y else plus (x-1) (y+1) in …
The Letrec Construct

- letrec \( x = E_1 \) in \( E_2 \)

**Attribute grammar rule**

\[
E ::= \text{letrec } x = E_1 \text{ in } E_2
\]

- Extensions over let rule
  1. \( T_{E_1} \) is inferred in augmented environment \( \Gamma_E + \{x:t_x\} \)
  2. Must unify \( S_{E_1}(t_x) \) and \( T_{E_1} \)
  3. Apply substitution \( S \) on top of \( S_{E_1} \)

Note: Can merge \text{let} and \text{letrec}, in \text{let}
\text{Unify} and \( S \) have no impact

\[
\begin{align*}
\Gamma_{E_1} &= \Gamma_E + \{x:t_x\} \\
S &= \text{Unify}(S_{E_1}(t_x), T_{E_1}) \\
\Gamma_{E_2} &= S \cdot S_{E_1}(\Gamma_E) + \{x:T_{E_1}\} \\
T_E &= T_{E_2} \\
S_E &= S_{E_2} \cdot S \cdot S_{E_1}
\end{align*}
\]
let vs. letrec

let \textbf{plus} = \lambda x. \lambda y. \text{if } (x=0) \text{ then } y \text{ else } ((\textbf{plus} \ x-1) \ y+1) \text{ in}

...
let/letrec Examples

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

- Typing `plus` using Strategy 1...
  \[ t_{plus} = t_x \rightarrow t_y \rightarrow t_1 \]
  \[ t_x = \text{int} \] // because of `x=0` and `x-1`
  \[ t_y = \text{int} \] // because of `y+1`
  Unify \( t_{plus}, \text{int} \rightarrow \text{int} \rightarrow \text{int} \) yields \( t_1 = \text{int} \)

- Haskell
  
  \[ \text{plus :: int -> int -> int} \]
  \[ \text{plus x y = if (x=0) then y else plus (x-1) (y+1)} \]
def W(\(\Gamma\), E) = case E of
  c  ->  (\[\], \text{TypeOf}(c))
  x  ->  if (x \text{ NOT in Dom}(\(\Gamma\))) then \text{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(\(S_{E1}(\Gamma), E_2\))
                 \(S = \text{Unify}(S_{E2}(T_{E1}), T_{E2}\rightarrow t)\)
                 in (\(S S_{E2} S_{E1}, S(t)\) // \(S S_{E2} S_{E1}\) composes substitutions)
  let \(x = E_1\) in \(E_2\) -> 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 = Γ(x);
                  in ([], T_E)
λx.E_1  ->  let (S_{E_1}, T_{E_1}) = W(Γ+{x:t},E_1)
            in (S_{E_1}, S_{E_1}(t_x)→T_{E_1})
E_1 E_2  ->  let (S_{E_1}, T_{E_1}) = W(Γ,E_1)
            (S_{E_2}, T_{E_2}) = W(S_{E_1}(Γ),E_2)
            S = Unify(S_{E_2}(T_{E_1}), T_{E_2}→t)
            in (S S_{E_2} S_{E_1}, S(t)) // S S_{E_2} S_{E_1} composes substitutions
let x = E_1 in E_2  ->  let (S_{E_1}, T_{E_1}) = W(Γ+{x:t},E_1)
                     S = Unify(S_{E_1}(t_x), T_{E_1})
                     (S_{E_2}, T_{E_2}) = W(S S_{E_1}(Γ)+{x:T_{E_1}},E_2)
                     in (S_{E_2} S S_{E_1}, T_{E_2})
Outline

- Simple type inference, cont.
  - Equality constraints, Unification, Substitution
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing
  - Let expressions

- Examples in context of HW6 starter code

- Hindley Milner: next week
Example: $\lambda f. \lambda x. (f (f x))$

- Creating constraints
Example: $\lambda f. \lambda x. (f (f x))$

- Solving constraints offline
Example: \((\lambda f. f \ 5) \ (\lambda x. \ x)\)