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 F₁)
  - 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

\[
\Gamma \vdash c : \text{int} \quad (\text{c is a constant})\\
\]

\[
\begin{align*}
x : \tau & \in \Gamma \\
\Gamma & \vdash x : \tau \\
\Gamma + \cdot x : \sigma & \\
\Gamma, x : \sigma & \vdash E_1 : \tau \\
\hline
\Gamma & \vdash (\lambda x : \sigma. E_1) : \sigma \to \tau \\
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash E_1 : \sigma \to \tau \\
\Gamma & \vdash E_2 : \sigma \\
\hline
\Gamma & \vdash (E_1 \ E_2) : \tau \\
\end{align*}
\]

Type expressions:
\[
\tau ::= \text{int} | \tau \to \tau
\]

Environment:
\[
\Gamma ::= \text{Nil} | \Gamma, x : \tau
\]
Extensions to Core Typing Rules

\[ \Gamma \vdash c : \text{int} \]

\[ \Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int} \]

\[ \Gamma \vdash E_1 + E_2 : \text{int} \]

\[ \Gamma \vdash E_1 = E_2 : \text{bool} \]

\[ \Gamma \vdash b : \text{bool} \quad \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau \]

\[ \Gamma \vdash \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau \]

(Comparison)
We can figure out all types even without explicit types for variables

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

Type inference

Type inference, Strategy 1

- Use typing rules to define type constraints
- Solve type constraints
- Aka constraint-based typing (e.g., Pierce)
\[ \text{Nil} \vdash (\lambda f. f \, 5) \, (\lambda x. x + 1) : \text{tf} \]

1. \text{App} \quad \Gamma = [f : \text{tf}] \\
\quad t_2 = t_4 \rightarrow t_2

2. \text{Abs} \quad \Gamma = [f : \text{tf}] \\
\quad t_2 = t_3 \rightarrow t_3

3. \text{App} \quad \Gamma = [f : \text{tf}] \\
\quad t_3 = t_3

4. \text{Abs} \quad \Gamma = [x : t_x] \\
\quad t_4 = \text{int} \rightarrow \text{int}

5. + \\
\quad t_5 = \text{int}

\[ \begin{align*}
\Gamma & = [f : \text{tf}, x : t_x] \\
\quad t_2 & = t_3 \rightarrow t_3 \\
\quad t_3 & = t_4 \\
\quad t_4 & = t_3 \\
\quad t_5 & = \text{int} \\
\quad t_x & = \text{int}
\end{align*} \]
### We Can Infer All Types!

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

1. **App**  
   \[ \Gamma = [\ ] \]
   \[ t_2 = t_4 \rightarrow t_1 \]

2. **Abs**  
   \[ \Gamma = [\ ] \]
   \[ t_2 = t_f \rightarrow t_3 \]

3. **App**  
   \[ \Gamma = [f:t_f] \]
   \[ t_f = \text{int} \rightarrow t_3 \]

4. **Abs**  
   \[ \Gamma = [x:t_x] \]
   \[ t_4 = t_x \rightarrow t_5 \]

5. **+**  
   \[ \Gamma = [x:t_x] \]
   \[ t_5 = \text{int} \]
   \[ t_x = \text{int} \]

\[ \Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int} \]

\[ \Gamma \vdash E_1 + E_2 : \text{int} \]

\[ \Gamma \vdash E_1 : \sigma \rightarrow \tau \quad \Gamma \vdash E_2 : \sigma \]

\[ \Gamma \vdash (E_1 \; E_2) : \tau \]

Program Analysis CSCI 4450/6450, A Milanova (Example term from MIT 2015 Program Analysis OCW)
We constructed a system of type constraints

Let’s solve the system of constraints

\[ t_2 = t_4 \rightarrow t_1 \]
\[ t_2 = t_f \rightarrow t_3 \]
\[ t_4 = t_x \rightarrow t_5 \]
\[ t_f = int \rightarrow t_3 \]
\[ t_5 = int, t_x = int \]

We inferred all \( t \)'s!

\[ t_1 = int \]
\[ t_2 = (int \rightarrow int) \rightarrow int \]
\[ t_3 = int \]
\[ t_4 = int \rightarrow int \]
\[ t_f = int \rightarrow int \]

\( (\lambda f: int \rightarrow int. \ f \ 5) \ (\lambda x: int. \ x + 1) : int \ (t_1) \)
Another Example

- \( \text{twice } f \ x = f \ (f \ x) \)
- What is the type of \texttt{twice}?
Another Example

- \texttt{twice f x = f (f x)}
- What is the type of \texttt{twice}?
  - It is $t_f \rightarrow t_x \rightarrow t_1$ ($t_1$ is the type of \texttt{f (f x)})
- Based on the syntax tree of \texttt{f (f x)} we have:
  
  \[
  t_f = t_2 \rightarrow t_1 \\
  t_f = t_x \rightarrow t_2
  \]

  Thus, $t_x = t_1 = t_2$, $t_f = t_x \rightarrow t_x$ and type of \texttt{twice} is $(t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x$

  Note: $t_x$ is a free type variable! Polymorphism!
Type Constraints from Typing Rules, as Attribute Grammar

Syntax: \( E ::= x \mid c \mid \lambda x.E \mid E_1 E_2 \mid E_1 + E_2 \)

Grammar rule:  
- \( E ::= x \)  
- \( E ::= c \)  
- \( E ::= \lambda x.E_1 \)  
- \( E ::= E_1 E_2 \)  
- \( E ::= E_1 + E_2 \)

Attribute rule:  
- \( C_E = \{ t_E = \Gamma_E(x) \} \)  
- \( C_E = \{ t_E = \text{int} \} \)  
- \( \Gamma_{E1} = \Gamma_E; x: t_x \)  
- \( C_E = C_{E1} \cup \{ t_E = t_x \rightarrow t_{E1} \} \)  
- \( \Gamma_{E1} = \Gamma_E \quad \Gamma_{E2} = \Gamma_E \)  
- \( C_E = C_{E1} \cup C_{E2} \cup \{ t_{E1} = t_{E2} \rightarrow t_E \} \)  
- \( \Gamma_{E1} = \Gamma_E \quad \Gamma_{E2} = \Gamma_E \)  
- \( C_E = C_{E1} \cup C_{E2} \cup \{ t_{E1} = \text{int}, t_{E2} = \text{int}, t_E = \text{int} \} \)
Type Constraints from Typing Rules, as Attribute Grammar

\[ E ::= \lambda x. E_1 \]

\[ \Gamma = \Gamma_{E_1} ; x : t_x \]

\[ C_E = C_{E_1} \cup \{ t_E = t_x \rightarrow t_{E_1} \} \]

\( t_E \) is “fresh” type variable for term represented by \( E \)’s subtree.

\[ E ::= E_1 E_2 \]

\[ \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \]

\[ C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = t_{E_2} \rightarrow t_E \} \ldots \]

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

\( \Gamma \) is inherited. Propagates top-down 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 $\tau_a$ and $\tau_b$ 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 $\tau_a = \text{int} \rightarrow t_1$
  $\tau_b = t_2 \rightarrow \text{bool}$
  - Can we unify $\tau_a$ and $\tau_b$? Yes, if $\text{bool}/t_1$ and $\text{int}/t_2$
- Suppose $\tau_a = \text{int} \rightarrow t_1$
  $\tau_b = \text{bool} \rightarrow \text{bool}$
  - Can we unify $\tau_a$ and $\tau_b$? No.
Example

\[ t_1 \rightarrow \text{bool} = (\text{int} \rightarrow t_2) \rightarrow t_3 \]

Yes, if \( \text{int} \rightarrow t_2/t_1 \) and \( \text{bool}/t_3 \)
Simple Type Substitution
(essential to define unification)

Language of types
\[ \tau ::= b \quad // \text{primitive type, e.g., int, bool} \]
| t \quad // \text{type variable} \]
| \tau \rightarrow \tau \quad // \text{function type} \]

A substitution is a map
- \( S : \text{Type Variable} \rightarrow \text{Type} \)
- \( S = [\tau_1/t_1, \ldots, \tau_n/t_n] \quad // \text{substitute type } \tau_i \text{ for type } \text{var } t_i \)

A substitution instance \( \tau' = S \tau \)
- \( S = [ t_0 \rightarrow \text{bool} / t_1 ] \quad \tau = t_1 \rightarrow t_1 \quad \text{then} \)
- \( S(\tau) = S(t_1 \rightarrow t_1) = (t_0 \rightarrow \text{bool}) \rightarrow (t_0 \rightarrow \text{bool}) \)
Substitutions can be composed

- $S_1 = [ \ t_0 \rightarrow \text{bool} / t_1 \ ]$
- $S_2 = [ \ \text{int} / t_0 \ ]$
- $\tau = t_1 \rightarrow t_1$
- $S_2 \ S_1 (\tau) = 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

- $S_1 = [ t_1 / t_2 ]$
- $S_2 = [ t_3 / t_1 ]$
- $S_3 = [ t_4 \rightarrow \text{int} / t_3 ]$

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

- A substitution $S_1$ is **less specific** (i.e., more general) than substitution $S_2$ if $S_2 = S \cdot S_1$ for some substitution $S$
  - E.g., $S_1 = [t_1 \mapsto t_1 / t_2]$ is more general than $S_2 = [\text{int} \mapsto \text{int} / t_2]$ because $S_2 = S \cdot S_1$ for $S = [\text{int} / t_1]$

- A **principal unifier** of a constraint set $C$ is a substitution $S_1$ that satisfies $C$, and $S_1$ is more general than any $S_2$ that satisfies $C$
Examples

- Find principal unifiers (when they exist) for

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

- $\text{Unify}(\text{int} \rightarrow \text{int}, t_1 \rightarrow t_2)$ yields ?

- $\text{Unify}(\text{int}, \text{int} \rightarrow t_2)$ yields ?

- $\text{Unify}(t_1, \text{int} \rightarrow t_2)$ 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 = { $\tau_1=\tau_2$ } U C'
        S = Unify ($\tau_1,\tau_2$) // Unify returns a substitution $S$
in
        UnifySet ( S(C') ) S
    // Compose the 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 \} 

- \{ 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)

- 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 $S$

**Grammar rule:**

- $E ::= x$
- $E ::= c$
- $E ::= \lambda x.E_1$
- $E ::= E_1 E_2$

**Attribute rule:**

- $T_E = \Gamma_E(x)$, $S_E = [ ]$
- $T_E = \text{int}$, $S_E = [ ]$
- $\Gamma_{E_1} = \Gamma_E ; x : t_x$
- $T_E = S_{E_1}(t_x) \rightarrow T_{E_1}$, $S_E = S_{E_1}$

$T_E$ 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.

$E_1$ $E_2$

- $\Gamma_{E_1} = \Gamma_E$
- $\Gamma_{E_2} = S_{E_1}(\Gamma_E)$
- $S = \text{Unify}(S_{E_2}(T_{E_1}), T_{E_2} \rightarrow t_E)$
- $T_E = S(t_E)$, $S_E = S S_{E_2} S_{E_1}$

Program Analysis CSCI 4450/6450, A Milanova 34
Example: \((\lambda f. f 5) (\lambda x. x)\)

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**
   - \(\Gamma_2 = []\)
   - \(T_2 = (\text{int} \rightarrow t_3) ightarrow t_3\)
   - \(S_2 = [\text{int} \rightarrow t_3/t_f]\)

3. **App**
   - \(\Gamma_3 = [f: t_f]\)
   - \(T_3 = t_3\)
   - \(S_3 = [\text{int} \rightarrow t_3/t_f]\)
   - \(\Gamma = [f: t_f]\)
   - \(T = t_f\)
   - \(S = []\)

4. **Abs**
   - \(\Gamma_4 = S_2(\Gamma_1) = []\)
   - \(T_4 = t_x \rightarrow t_x\)
   - \(S_4 = []\)
   - \(\Gamma = [x: t_x]\)
   - \(T = t_x\)
   - \(S = []\)

Steps at 1, finally:
1. unify( (\text{int} \rightarrow t_3) \rightarrow t_3, (t_x \rightarrow t_x) \rightarrow t_1 )
   returns \(S = [\text{int}/t_x, \text{int}/t_3, \text{int}/t_1]\)
2. \(S_1 = S\ S_4\ S_2 = S\ S_2 = S\ [\text{int} \rightarrow t_3/t_f]\)
3. \(T_1 = S(t_1) = \text{int}\)
Example: $\lambda f. \lambda x. (f (f \ x))$
The Let Construct

- In dynamic semantics, \( \text{let } x = E_1 \text{ in } E_2 \) is equivalent to \( (\lambda x. E_2) \ E_1 \)

- Typing rule

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

- In static semantics \( \text{let } x = E_1 \text{ in } E_2 \) is not equivalent to \( (\lambda x. E_2) \ E_1 \)

  - In \textit{let}, the type of “argument” \( E_1 \) is inferred/checked \textbf{before} the type of function body \( E_2 \)
  
  - \textit{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 \( x = E_1 \) in \( E_2 \)
  - \( x \) can be referenced from within \( E_1 \)
  - Extends calculus with general recursion
    - No need to type \texttt{fix} (we can’t!) but we can still type recursive functions like \texttt{plus}, \texttt{times}, etc.
  - Haskell’s \texttt{let} is a \texttt{letrec} actually…

- E.g.,
  \[
  \text{letrec plus} = \lambda x.\lambda y. \begin{cases} 
  y & \text{if } (x=0) \\
  \text{plus} (x-1) (y+1) & \text{else}
  \end{cases}
  \]
  written as
  \[
  \text{letrec plus } x \ y = \begin{cases} 
  y & \text{if } (x=0) \\
  \text{plus} \ (x-1) \ (y+1) & \text{else}
  \end{cases}
  \]
The Letrec Construct

- **letrec x = E₁ in E₂**

### Attribute grammar rule

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

- \( \Gamma_{E_1} = \Gamma_E + \{x: t_x\} \)
- \( S = \text{Unify}(S_{E_1}(t_x), T_{E_1}) \)
- \( \Gamma_{E_2} = S \cdot S_{E_1}(\Gamma_E) + \{x: T_{E_1}\} \)
- \( T_E = T_{E_2} \quad S_E = S_{E_2} \cdot S \cdot S_{E_1} \)

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 **let** and **letrec**, in **let**

**Unify** and **S** have no impact
let/letrec Examples

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

Typing plus using Strategy 1...

\[ t_{\text{plus}} = t_x \rightarrow t_y \rightarrow t_1 \]

\[ t_x = \text{int} \] // because of x=0 and x-1

\[ t_y = \text{int} \] // because of y+1

Unify(\( t_{\text{plus}} \), \( \text{int} \rightarrow \text{int} \rightarrow \text{int} \)) yields \( t_1 = \text{int} \)

Haskell

\[ \text{plus :: int} \rightarrow \text{int} \rightarrow \text{int} \]

\[ \text{plus} \ x \ y = \text{if } (x=0) \text{ then } y \text{ else } \text{plus} \ (x-1) \ (y+1) \]
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_E1, T_E1) = W(Γ+{x:t_x},E_1)
                in (S_E1, S_E1(t_x)→T_E1)
    E_1 E_2 -> let (S_E1, T_E1) = W(Γ,E_1)
                (S_E2, T_E2) = W(S_E1(Γ),E_2)
                S = Unify(S_E2(T_E1),T_E2→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(Γ,E_1)
                        (S_E2, T_E2) = W(S_E1(Γ)+{x:T_E1},E_2)
                        in (S_E2 S_E1, T_E2)
Algorithm W, Almost There! (merges let and letrec)

```python
def W(Γ, E) = case E of
    c   -> ([], TypeOf(c))
    x   -> if (x NOT in Dom(Γ)) then fail
         else let TE = Γ(x); in ([], TE)
    λx.E1 -> let (SE1, TE1) = W(Γ+{x:t}, E1) in (SE1, SE1(t)→TE1)
    E1 E2 -> let (SE1, TE1) = W(Γ, E1)
             (SE2, TE2) = W(Γ, E2)
             S = Unify(SE2(TE1), TE2→t)
             in (S SE2 SE1, S(t)) // S SE2 SE1 composes substitutions
    let x = E1 in E2 -> let (SE1, TE1) = W(Γ+{x:t}, E1)
                         S = Unify(SE1(t), TE1)
                         (SE2, TE2) = W(S SE1(Γ)+{x:T}, E2)
                         in (SE2 S SE1, TE2)
```
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