Simple Type Inference
Announcements

- HW6: Pushed back deadline
- Quiz 5: Pushed back for Friday

- Paper presentation guidelines are up
- Papers coming up
Last Week

- 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
- Intro to simple type inference
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
Static Semantics

(Variable)

$$\frac{x : \tau \in \Gamma}{\Gamma |- x : \tau}$$

(Application)

$$\frac{\Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma}{\Gamma |- (E_1 E_2) : \tau}$$

(binding) augments environment $\Gamma$

$$\frac{\Gamma, x : \sigma |- E_1 : \tau}{\Gamma |- (\lambda x : \sigma. E_1) : \sigma \rightarrow \tau}$$

(Variable)

looks up the type of $x$ in environment $\Gamma$

x:τ ∈ Γ

Γ |- x : τ

Γ |- E₁ : σ → τ  Γ |- E₂ : σ

Γ |- (E₁ E₂) : τ

binding: augments environment $\Gamma$

Γ, x : σ |- E₁ : τ

Γ |- (λx : σ. E₁) : σ → τ

Γ |- (E₁ E₂) : τ

Γ |- (λx : σ. E₁) : σ → τ
Deducing Types

\[ \lambda x: \text{int}. \lambda y: \text{bool}. \ x \]

1. Abs  \( \Gamma = [] \)
   \[ t_1 = \text{int} \rightarrow \text{bool} \rightarrow \text{int} \]

2. Abs  \( \Gamma = [x: \text{int}] \)
   \[ t_2 = \text{bool} \rightarrow \text{int} \]

3. Var  \( x \)  \( \Gamma = [x: \text{int}, y: \text{bool}] \)
   \[ t_3 = \text{int} \]

1,2,3 denote the subcomponents of the term. We will be deducing types for each of these components.
Deducing Types

$(\lambda f: \text{int} \rightarrow \text{int}. \ f \ 5) \ (\lambda x: \text{int}. \ x + 1) : ?$
Type Inference, Strategy 1

- 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)
We Can Infer All Types!

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

1. App  \hspace{1cm} \Gamma = []

2. Abs  \hspace{1cm} \Gamma = []

3. App  \hspace{1cm} \Gamma = [f:t_f]

\[ t_f = \text{int} \rightarrow t_3 \]

\(\lambda f: t_f \)

\[ \Gamma = [f:t_f] \]

4. Abs  \hspace{1cm} \Gamma = [x:t_x]

\[ t_4 = t_x \rightarrow t_5 \]

\(\lambda x: t_x \)

\[ \Gamma = [x:t_x] \]

5. +  \hspace{1cm} \Gamma = [x:t_x]

\[ t_5 = \text{int} \]

\[ t_x = \text{int} \]

\(\text{Var } f \)

\(\text{Const } 5\)

\(\Gamma = [] \)

\(\text{Var } x \)

\(\text{Const } 1\)

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

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

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

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

\[ \Gamma \vdash E_2 : \sigma \]

\[ \Gamma \vdash (E_1 \, E_2) : \tau \]
We constructed a system of type constraints

Let’s solve the system of constraints

\[ t_2 = \text{int} \rightarrow t_1 \quad t_f = \text{int} \rightarrow t_3 \quad t_4 = \text{int} \rightarrow \text{int} \]
\[ t_2 = t_f \rightarrow t_3 \quad t_3 = \text{int} \quad t_1 = t_3 = \text{int} \]
\[ t_4 = t_x \rightarrow t_5 \quad t_4 = \text{int} \rightarrow \text{int} \]
\[ t_f = \text{int} \rightarrow t_3 \quad t_5 = \text{int}, \quad t_x = \text{int} \]

We inferred all \( t \)'s!

\[ t_1 = \text{int} \quad t_2 = (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \]
\[ t_3 = \text{int} \quad t_4 = \text{int} \rightarrow \text{int} \]
\[ t_f = \text{int} \rightarrow \text{int} \]

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

- \texttt{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 \( f \, (f \, x) \))
- Based on the syntax tree of \( f \, (f \, x) \) 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! \textbf{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: Attribute rule:

- \( E ::= x \)
  - \( C_E = \{ t_E = \Gamma_E(x) \} \)
- \( E ::= c \)
  - \( C_E = \{ t_E = \text{int} \} \)
- \( E ::= \lambda x.E_1 \)
  - \( \Gamma_{E_1} = \Gamma_E; x:t_X \)
  - \( C_E = C_{E_1} \cup \{ t_E = t_X \rightarrow t_{E_1} \} \)
- \( E ::= E_1 E_2 \)
  - \( \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \)
  - \( C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = t_{E_2} \rightarrow t_E \} \)
- \( E ::= E_1 + E_2 \)
  - \( \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \)
  - \( C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = \text{int}, t_{E_2} = \text{int}, t_E = \text{int} \} \)
Type Constraints from Typing Rules, as Attribute Grammar

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

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

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

\[ E ::= E_1 E_2 \]

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

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

\( \Gamma \) is inherited. Propagates top-down the tree.

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

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

\( \lambda f. \lambda x. f (f \ x) \)
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 $\tau_a = \begin{array}{c} t_1 \rightarrow t_2 \\ \tau_b = \begin{array}{c} t_3 \rightarrow t_4 \end{array} \end{array}$

- Structural equality entails $\tau_a = \tau_b \text{ means } t_1 \rightarrow t_2 = t_3 \rightarrow t_4 \text{ iff } t_1 = t_3 \text{ and } t_2 = t_4$
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
  \[ \tau ::= b \quad \text{// primitive type, e.g., int, bool} \]
  \[ | t \quad \text{// type variable} \]
  \[ | \tau \to \tau \quad \text{// function type} \]

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

- A substitution instance \( \tau' = S \tau \)
  - \( S = [t_0 \to \text{bool} / t_1] \quad \tau = t_1 \to t_1 \) then
  - \( S(\tau) = S(t_1 \to t_1) = (t_0 \to \text{bool}) \to (t_0 \to \text{bool}) \)
Simple Type Substitution
(essential to define unification)

- Substitutions can be composed
  - $S_1 = [ t_0 \mapsto \text{bool} / t_1 ]$
  - $S_2 = [ \text{int} / t_0 ]$
  - $\tau = t_1 \mapsto t_1$
  - $S_2 S_1 (\tau) = S_2 ( S_1 (t_1 \mapsto t_1) ) =$
    
    $S_2 ( (t_0 \mapsto \text{bool}) \mapsto (t_0 \mapsto \text{bool}) ) =$
    
    $(\text{int} \mapsto \text{bool}) \mapsto (\text{int} \mapsto \text{bool})$
Examples

Substitutions can be composed

- $S_1 = [ \frac{t_x}{t_1} ]$
- $S_2 = [ \frac{t_x}{t_2} ]$

- $\tau = t_2 \rightarrow t_1$
- $S_2 S_1(\tau) = ?$
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., $S_1 = [ t_1 \rightarrow t_1 / t_2 ]$ is more general than $S_2 = [ \text{int} \rightarrow \text{int} / t_2 ]$ 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 $C$. 

Program Analysis CSCI 4450/6450, A Milanova 26
Examples

- Find principal unifiers (when they exist) for
  - \{ \text{int} \rightarrow \text{int} = t_1 \rightarrow t_2 \}\n  - \{ \text{int} = \text{int} \rightarrow t_2 \}\n  - \{ t_1 = \text{int} \rightarrow t_2 \}\n  - \{ t_1 = \text{int}, t_2 = t_1 \rightarrow t_1 \}\n  - \{ 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(τ₁, τ₂) =
    case (τ₁, τ₂)
    (τ₁, t₂) = [τ₁/t₂] provided t₂ does not occur in τ₁
    (t₁, τ₂) = [τ₂/t₁] provided t₁ does not occur in τ₂
    (b₁, b₂) = if (eq? b₁ b₂) then [ ] else fail
    (τ₁₁→τ₁₂, τ₂₁→τ₂₂) = let
        S₁ = Unify(τ₁₁, τ₂₁)
        S₂ = Unify(S₁(τ₁₂), S₁(τ₂₂))
    in S₂ S₁ // compose substitutions
otherwise = fail
```

This is the occurs check!
Examples

- Unify (int → int, t₁ → t₂) yields ?

- Unify (int, int → t₂) yields ?

- Unify (t₁, int → t₂) 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
    // 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 \} \\

- \{ 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 slides 14-15)

- Solve type constraints offline
  - Use unification algorithm
  - (Pseudo code in slide 30)
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
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:**

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

**Attribute rule:**

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

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

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

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. **\(S_1 = S\)**
   - **\(S_2 = S\)**
   - **\(S_2 = S\)**
   - **\(S_2 = [\text{int} \rightarrow t_3 / t_f]\)**
3. **\(T_1 = S(t_1) = \text{int}\)**
   - from Unify(\(t_f, \text{int} \rightarrow t_3\))
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
  \[
  \frac{
  \Gamma |- E_1 : \sigma \\
  \Gamma; x: \sigma |- E_2 : \tau
  }{
  \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau
  }
  \]

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

\[ \Gamma |- E_1 : \sigma \quad \Gamma ; x : \sigma |- E_2 : \tau \]

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

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 `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` = $\lambda x.\lambda y. \text{if } (x=0) \text{ then } y \text{ else } ((\text{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 \( E_2 \)

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

**Unify** and **S** have no impact

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

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

- **Typing** **plus** using Strategy 1...

  \[ t_{\text{plus}} = t_x \rightarrow t_y \rightarrow t_1 \]
  \[ t_x = \text{int} \quad // \text{because of } x=0 \text{ and } x-1 \]
  \[ t_y = \text{int} \quad // \text{because of } y+1 \]

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

- **Haskell**

  **plus** :: int -> int -> int
  **plus** x y = if (x=0) then y else **plus** (x-1) (y+1)
Algorithm W, Almost There!

```python
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_x}, 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,E_1)
            (S_{E_2}, T_{E_2}) = W(S_{E_1}(Γ)+{x:T_{E_1}}, E_2)
            in (S_{E_2}, S_{E_1}, T_{E_2})
```

Program Analysis CSCI 4450/6450, A Milanova (from MIT 2015 Program Analysis OCW)
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_x}, 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_x}, 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
  - 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