Simple Type Inference, continued

Announcements
- HW5 due today
  - How is it going?
- HW6 is up on Submitty
  - Team assignment
- Paper presentation guidelines are up
- Papers coming up

Outline
- Simple type inference
  - Equality constraints
  - Unification
    - Structural equality, substitution, principal unifiers
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism
- 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

Type Inference, Strategy 1
- Type inference: deduces types for variables and terms without explicit type annotations on variables (or with only partial annotations)
  - E.g., \((\lambda f\ .\ f\ 5)\ (\lambda x\ .\ x+1) : ?\)
- Type inference, Strategy 1
  - First, derive type constraints from type rules
  - Second, solve the 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

$t_2 = t_4 \rightarrow t_1$
$t_2 = t_6 \rightarrow t_3$
$t_4 = t_2 \rightarrow t_5$
$t_4 = int \rightarrow t_3$
$t_5 = int, t_2 = int$

$(\lambda f : int \rightarrow int. f \, 5) \, (\lambda x : int. x+1) : int \langle t_1 \rangle$

Solving Constraints
- Two key concepts
  - Equality
  - Unification

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 $t_1 = t_2 = t_3 = t_4$

Equality and Unification
- Can two types be made equal by choosing appropriate substitutions for their type variables?
- Robinson's unification algorithm

Example
- $t_1 \rightarrow bool = (int \rightarrow t_2) \rightarrow t_3$
  
  ![Diagram]

  Yes, if $int \rightarrow t_2 / t_1$ and $bool / t_3$
Examples

- Substitutions can be composed
  - \( S_1 = \{ t_1 \rightarrow t_2 \} \)
  - \( S_2 = \{ t_3 \rightarrow t_1 \} \)
  - \( S_3 = \{ t_4 \rightarrow \text{int} \rightarrow t_5 \} \)
  - \( \tau = t_1 \rightarrow t_2 \)
  - \( S_2 S_1(\tau) = ? \)
  - \( S_2 S_3(t_4 \rightarrow \text{int} \rightarrow t_5) = \)
  - \( (t_4 \rightarrow \text{int} \rightarrow t_5) \rightarrow t_5 \rightarrow \text{int} \)

More Terminology...

- A substitution \( S_1 \) is less specific (i.e., more general) than substitution \( S_2 \) if \( S_2 = S S_1 \) for some substitution \( S \)
  - E.g., \( S_1 = \{ t_1 \rightarrow t_4 \} \) is more general than \( S_2 = \{ \text{int} \rightarrow \text{int} \rightarrow t_2 \} \) because \( S_2 = S S_1 \) for \( S = \{ \text{int} \rightarrow t_1 \} \)
- A principal unifier of a constraint set \( C \) is a substitution \( S_1 \) that satisfies \( C \), and \( S_1 \) is more general than any \( S_2 \) satisfying \( C \)

Examples

- Find principal unifiers (when they exist) for
  - \( \{ \text{int} \rightarrow \text{int} \rightarrow t_1 \rightarrow t_2 \} \)
  - \( \{ \text{int} \rightarrow t_2 \} \)
  - \( \{ t_1 \rightarrow \text{int} \rightarrow t_2 \} \)
  - \( \{ t_1 \rightarrow \text{int} \rightarrow t_1 \} \)
  - \( \{ t_1 \rightarrow t_2 = t_2 \rightarrow t_1 \} \)
  - \( \{ t_1 \rightarrow t_2 = t_2 \rightarrow t_3, t_3 = t_4 \rightarrow t_5 \} \)

Unification

(essential for type inference!)

- \( \text{Unify: tries to unify } \tau_1 \text{ and } \tau_2 \text{; returns a principal unifier for } \tau_1 = \tau_2 \text{ if unification is successful} \)
  - \( \text{Unify}(\tau_1, \tau_2) = \)
    - case \((\tau_1, \tau_2)\)
      - \((\tau_1, \tau_2) = [\tau_1/t_2] \text{ provided } t_2 \text{ does not occur in } \tau_1\)
      - \((\tau_1, \tau_2) = [\tau_2/t_1] \text{ provided } t_1 \text{ does not occur in } \tau_2\)
      - \((b_1, b_2) = \text{if } (\text{eq? } b_1, b_2) \text{ then } [ ] \text{ else fail} \)
      - \((\tau_1 \rightarrow \tau_2, \tau_2 \rightarrow \tau_3) = \text{let } S_1 = \text{Unify}(\tau_1, \tau_2) \)
      - \(S_2 = \text{Unify}(S_1, \tau_2) S_1(\tau_3) \)
      - in \( S_2 S_1 \) // compose substitutions
  - otherwise = \text{fail}

Examples

- \( \text{Unify } (\text{int} \rightarrow \text{int}, t_1 \rightarrow t_2) \text{ yields?} \)
  - \( [\text{int} \rightarrow t_2/\text{int} \rightarrow t_1] \)
- \( \text{Unify } (\text{int}, \text{int} \rightarrow t_2) \text{ yields?} \)
  - \( \text{fail} \)
- \( \text{Unify } (t_1, \text{int} \rightarrow t_2) \text{ yields?} \)
  - \( [\text{int} \rightarrow t_2/\text{int} \rightarrow t_1] \)

Unify Set of Constraints \( C \)

- \( \text{UnifySet: tries to unify } C \text{ and returns a principal unifier for } C \text{ if unification is successful} \)
  - \( \text{UnifySet } (C) = \)
    - if \( C \) is Empty Set then \([ ]\)
    - else let
      - \( C = \{ \tau_1 = \tau_2 \} U C' \)
      - \( S = \text{Unify}(\tau_1, \tau_2) \)
      - in
        - \( \text{UnifySet } (S(C')) * S \)
        - // Composition of substitutions
Examples

- \{ t_1 = \text{int}, t_2 = t_1 \to t_3 \}\n- \{ t_1 \to t_2 = t_2 \to t_3, t_3 = t_2 \to t_5 \}\n- \{ t_2 = t_2 \to t_1, t_2 = t_1 \to t_2 \}\n- \{ t_3 = t_1 \to t_1, t_4 = t_2 \to t_2, t = \text{int} \to t_2, t_2 = \text{int} \to t_3, t_2 = \text{int} \to \text{int} \}\n
Outline

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

Replace Attribute C (Constraints) with Attribute S (Substitution Map)

Grammar rule:

| E ::= x     | \( T_E = \Gamma_E(x) \) | \( S_E = [ ] \) |
| E ::= c     | \( T_E = \text{int} \) | \( S_E = [ ] \) |
| E ::= \lambda x.E_1 | \( \Gamma_{E_1} = \Gamma_E \times t_2 \) | \( T_E = S_{E_1}(t_2) \to T_{E_1} \) | \( S_E = S_{E_1} \) |
| E ::= E_1 E_2 | \( \Gamma_{E_1} = \Gamma_E \) | \( \Gamma_{E_2} = S_{E_1}(\Gamma_E) \) | \( S = \text{Unify}(S_{E_2}(T_{E_1}), T_{E_2} \to t_3) \) | \( T_E = S(t_2) \to t_3 \) | \( S_E = S_{E_2} S_{E_1} \) |

Type Inference, Strategy 2

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

Example: \((\lambda f. f 5) \ (\lambda x. x)\)

\[
\begin{aligned}
&\begin{array}{c}
1. \text{App} \\
&\Gamma = [ ] \\
&S_1 = [ ] \\
&T_2 = (\text{int} \to \text{int}) \to \text{int} \\
&S_2 = [ \text{int} \to \text{int}/t_1 ] \\
\end{array} \\
&\begin{array}{c}
2. \text{Abs} \\
&\Gamma_1 = [ ] \\
&S_1 = [ ] \\
&T_2 = (\text{int} \to \text{int}) \to \text{int} \\
&S_2 = [ \text{int} \to \text{int}/t_1 ] \\
\end{array} \\
&\begin{array}{c}
3. \text{App} \\
&\Gamma = [f:t_1] \\
&S_1 = [ ] \\
&T_2 = t_1 \\
&S_2 = [ \text{int} \to \text{int}/t_1 ] \\
\end{array} \\
&\begin{array}{c}
4. \text{Abs} \\
&\Gamma = [x:t_2] \\
&S_1 = [ ] \\
&T_2 = t_2 \\
&S_2 = [ ] \\
\end{array} \\
&\begin{array}{c}
\lambda f. t_1 \\
\lambda x. t_2 \\
\text{Var } x \\
T = t_2 \\
S = [ ] \\
\end{array} \\
&\begin{array}{c}
\text{Const } 5 = t_1 \\
S = [ ] \\
\end{array} \\
&\text{from } \text{Unify}(t_1, t_1) \\
\end{aligned}
\]

Example: \((\lambda f. \lambda x. (f \ (f \ x)))\)
The Let Construct

- In dynamic semantics, let \( x = E_1 \) in \( E_2 \) is equivalent to \( (\lambda x.E_2) \ E_1 \)
- Typing rule
  \[
  \Gamma \vdash E_1 : \sigma \quad \Gamma, x : \sigma \vdash E_2 : \tau \\
  \Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau
  \]
- In static semantics let \( x = E_1 \) 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 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 \( \text{plus}, \text{times} \), etc.
  - Haskell’s let is a letrec actually...
- E.g.,
  - letrec \( \text{plus} = \lambda x.y. \text{if } (x=0) \text{ then } y \text{ else } (x-1) \ (y+1) \)
    written as
  - letrec \( \text{plus } x \ y = \text{if } (x=0) \text{ then } y \text{ else } \text{plus } (x-1) \ (y+1) \)

let/letrec Examples

letrec \( \text{plus } x \ y = \text{if } (x=0) \text{ then } y \text{ else } \text{plus } (x-1) \ (y+1) \)

- Typing \( \text{plus} \) using Strategy 1...
  \[
  t_{\text{plus}} = t_1 \rightarrow t_2 \rightarrow t_1 \\
  t_2 = \text{int} // \text{because of } x=0 \text{ and } x-1 \\
  t_1 = \text{int} // \text{because of } y+1
  \]
  \( t_1 \) is type of term \( \text{plus } (x-1) \ (y+1) \)
  Unify(\( t_{\text{plus}} \cdot \text{int} \rightarrow \text{int} \rightarrow \text{int} \)) yields \( t_1 = \text{int} \)
- Haskell
  - plus :: int -> int -> int
    plus \( x \ y = \text{if } (x=0) \text{ then } y \text{ else plus } (x-1) \ (y+1) \)

Algorithm \( W \), Almost There!

def \( W(\tau, E) \) = case \( E \) of
  c  \rightarrow (\tau, \text{TypeOf}(c))
  _x \rightarrow \text{if } (x \text{ NOT in } \text{Dom}(\tau)) \text{ then fail }
  \text{ else let } \tau_2 = \text{ Union } (x: \tau_1);
  \text{ in } (\tau_1, \tau_2)
  \lambda x. E_1 \rightarrow \text{let } (S_b, t_{\text{def}}) = \text{ W}(\tau(x), E_1)
  \text{ in } (S_b, t_{\text{def}} \rightarrow t_{\text{def}})
  E_1 \rightarrow \text{let } (S_b, t_{\text{def}}) = \text{ W}(\tau, E_1)
  (S_b, t_{\text{def}} \rightarrow W(S_b, t_{\text{def}}), E_2)
  S = \text{ Unif}(S_b, t_{\text{def}} \rightarrow E_2)
  \text{ in } (S, S_{\text{def}}, S_{\text{def}})

let \( x = E_1 \) in \( E_2 \)
  \text{let } (S_b, t_{\text{def}}) = \text{ W}(\tau, E_1)
  (S_b, t_{\text{def}} \rightarrow W(S_b, t_{\text{def}}), E_2)
  \text{ in } (S_b, S_{\text{def}}, T_{\text{def}})
Algorithm W, Almost There! (merges let and letrec)

\[
\text{def } \text{W}(\tau, E) = \text{case } E \text{ of}\]
\[c \rightarrow \text{((}) \cdot \text{TypeOf}(c))\]
\[x \rightarrow \text{if } (x \text{ NOT in } \text{Dom}(\tau)) \text{ then fail}\]
\[\text{else let } \tau = \Gamma(x);\]
\[\text{in } ((\tau, \text{Dom}(\tau)))\]
\[\lambda x. E_1 \rightarrow \text{let } (S_0, T_0) = \text{W}(\Gamma(x), E_1)\]
\[\text{in } (S_0, S_0, T_0, T_0)\]
\[E_2 \rightarrow \text{let } (S_1, T_1) = \text{W}(\Gamma(x), E_2)\]
\[\text{let } (S_2, T_2) = \text{W}(\Gamma(x), E_2)\]
\[\text{let } (S_3, T_3) = \text{W}(\Gamma(x), E_2)\]
\[\text{in } (S_3, S_3, T_3)\]

W is Standard Recursive Descend

\[\text{W}(\lambda, E) =\]
\[\text{case } E \text{ of}\]
\[\text{App } E_1, E_2 \rightarrow \text{let}\]
\[s_1 = \text{W}(\lambda, E_1)\]
\[\ldots\]
\[s_2 = \text{W}(\lambda, E_2)\]
\[\text{in}\]
\[s = g(s_1, s_2)\]

Motivating Example

- A sound type system rejects some programs that don’t get stuck
- Canonical example:
  \[\text{let } f = \lambda x. x\]
  \[\text{in}\]
  \[\text{if } (f \text{ true}) \text{ then } (f 1) \text{ else } 1\]
- Term does not get “stuck”
- Term is not typable in the simply typed lambda calculus. But it is typable in Hindley Milner

Different Styles of (Parametric) Polymorphism

- Impredicative polymorphism \((\text{System F})\)
  \[\tau ::= \forall \tau \cdot T \mid T \mid \forall \tau \cdot T\]
  \[E ::= x | \lambda x : \tau \cdot E | E_1 E_2 | \land E | E [\tau]\]

- Very powerful
  - Can type self application \(\lambda x. x x\)
  - Still cannot type fix!

- Type inference is undecidable!

The Polymorphic Lambda Calculus (\(\text{System F}\))

- System F adds two rules to System \(F_1\)
  - Dynamic semantics
    \[E_1 \rightarrow E_2\]
    \[\Gamma, E_1 \vdash E_2\]
    \[\Gamma, \tau \vdash E_\tau\]
    \[\Gamma, \tau \vdash E_\tau\]
  - Static semantics
    \[\Gamma, \tau \vdash E\]
    \[\Gamma, \tau \vdash E\]
    \[\Gamma, \tau \vdash E\]

- E.g., \((\forall \tau. \lambda x : \tau \cdot x [\text{int}]) 1 \rightarrow (\lambda x : \text{int} \cdot x) 1 \rightarrow 1\)
Different Styles of Polymorphism

- Predicative polymorphism
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall \tau \cdot \sigma \mid \sigma_1 \rightarrow \sigma_2 \]
  \[ E ::= x \mid \lambda x : \tau . E \mid E_1 E_2 \mid \Lambda T.E \mid E [\tau] \]

- Still very powerful
  - Restricts System F by disallowing instantiation with a polymorphic type: \( E [\tau] \) but not \( E [\sigma] \)
  - Type inference is still undecidable!

- Let polymorphism
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall \tau \cdot \sigma \]
  \[ E ::= x \mid \lambda x : \tau . E \mid E_1 E_2 \mid \Lambda T.E \mid E [\tau] \mid \text{let } x = E_1 \text{ in } E_2 \]

  - Like \( (\lambda x . E_2) \) but \( x \) can be polymorphic!
  - Good engineering compromise
    - Enhance expressiveness
    - Preserve decidability
  - This is the Hindley Milner type system

Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Simple type inference on-the-fly
  - Algorithm W, almost

- Parametric polymorphism
- Hindley Milner type inference