Announcements

- HW5 due today
- HW6 coming up tonight
- Paper presentation guidelines are up
- Papers coming up
- No class on Monday March 25th

Simple Type Inference

Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism (next time…)
- Hindley Milner type inference. Algorithm W

Last Class

- Introduction to types and type systems
- Simply typed lambda calculus, as known as System $F_1$
  - Language syntax, type expression syntax
  - Static semantics
  - Dynamic semantics
  - Type soundness: Safety = Progress + Preservation
  - Proved for the simply typed lambda calculus

Reading

- “Types and Programming Languages”, by Benjamin Pierce, Chapter 22, 23
- Lecture notes based partially on MIT 2015 Program Analysis OCW

Static Semantics

- $\Gamma \vdash x : \tau$ (Variable)
- $\Gamma \vdash E_1 : \sigma \rightarrow \tau$, $\Gamma \vdash E_2 : \sigma \rightarrow \tau$
  - $\Gamma \vdash (E_1 E_2) : \tau$ (Application)
- $\Gamma, x : \tau \vdash E_1 : \tau$
  - $\Gamma, (\lambda x : \sigma E_1) : \sigma \rightarrow \tau$ (Abstraction)
Deducing Types

1. Abs $\Gamma = []$
   $t_1 = \text{int}$
   $t_2 = \text{int}$
   $t_3 = \text{int}$

2. Abs $\Gamma = [x:\text{int}]$
   $t_2 = \text{int}$
   $t_3 = \text{int}$
   $t_4 = \text{int}$

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.

Type Inference, Strategy 1

- We can figure out all types even without explicit types for variables
  - $(\lambda f. f) (\lambda x. x+1) : ?$
- 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!

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

Type Constraints

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

We inferred all $t$'s:

$\begin{align*}
    t_2 &= t_4 \\
    t_2 &= t_7 \\
    t_3 &= t_7 \\
    t_4 &= t_7 \\
    t_7 &= t_7 \\
    t_3 &= t_1 \\
    t_5 &= t_1
\end{align*}$

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

Another Example

- twice $f x = f (f x)$
- What is the type of twice?
  - It is $t_1 \rightarrow t_5 \rightarrow t_1$ ($t_1$ is the type of $f (f x)$)
  - Based on the syntax tree of $f (f x)$ we have:
    
    $\begin{align*}
    t_2 &= t_3 \\
    t_3 &= t_4 \\
    t_4 &= t_1 \\
    t_5 &= t_1
    \end{align*}$
  
  Thus, $t_2 = t_1 = t_2$. $t_1 = t_5$. $t_7 = t_1$. and type of twice is $(t_1 \to t_3) \to t_5 \to t_1$.

Note: $t_5$ is a free type variable! Polymorphism!
Type Constraints from Typing Rules, as Attribute Grammar

- **Syntax:**
  
  \[ E ::= x \mid c \mid \lambda x. E \mid E_1 \cdot E_2 \mid E_1 + E_2 \]

- **Grammar rule:**
  
  \[ C_E = \{ t_E = \Gamma_E(x) \} \]

- **Attribute rule:**
  
  \[ \Gamma_E = \Gamma_E(x) ; x : t_E \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E ::= \lambda x. E_1 ]</td>
<td>[ \Gamma_E = \Gamma_E(x) ; x : t_E ]</td>
<td>[ \Gamma_E = \Gamma_E(x) \cup C_{E_1} \cup { t_{E_1} = t_{E_2} } ]</td>
<td></td>
</tr>
<tr>
<td>[ E ::= E_1, E_2 ]</td>
<td>[ \Gamma_E = \Gamma_E(x) \cup C_{E_1} \cup C_{E_2} \cup { t_{E_1} = t_{E_2} } ]</td>
<td>[ \Gamma_E = \Gamma_E(x) \cup C_{E_1} \cup C_{E_2} \cup { t_{E_1} = t_{E_2} } ]</td>
<td></td>
</tr>
<tr>
<td>[ E ::= E_1 + E_2 ]</td>
<td>[ \Gamma_E = \Gamma_E(x) \cup C_{E_1} \cup C_{E_2} \cup { t_{E_1} = int, t_{E_2} = int, t_E = int } ]</td>
<td>[ \Gamma_E = \Gamma_E(x) \cup C_{E_1} \cup C_{E_2} \cup { t_{E_1} = t_{E_2} } ]</td>
<td></td>
</tr>
</tbody>
</table>

**Example**

\[ \lambda f. \lambda x. f(fx) \]

**Standard Way of Writing This...**

- Semantic rules over syntax, generate constraints, i.e., attribute grammar!
- E.g., rule for abstraction \( A \)

\[ \begin{align*}
| \Gamma |- \lambda x. E_1 : t & \iff \exists t_{E_1} : a \cdot ((| \Gamma ; x : t \Gamma |- E_1 : a |) \\
& \land t = t_{E_2} \Rightarrow a)
\end{align*} \]

This reads: Constraints for abstraction term \( A \) given environment \( \Gamma \) include all constraints generated for term \( E_1 \) given augmented environment \( \Gamma ; x : t \) and constraint \( t = t_{E_2} \Rightarrow a \), for term \( A \) itself. \( t_{E_1} \) and \( a \) are fresh type variables created along derivation.

**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 \( t_1 \Rightarrow t_2 \land t_3 \Rightarrow t_4 \) iff \( t_1 = t_3 \) and \( t_2 = t_4 \)

---

*Spring 19 CSCI 4450/6450, A Milanova*
Equality and Unification

- Can two types be made equal by choosing appropriate substitutions for their type variables?
- Robinson’s unification algorithm
  - Suppose \( \tau_a = \text{int} \rightarrow t_1 \)
  - Can we unify \( \tau_a \) and \( \tau_b \)? Yes, if \( \text{bool} / t_1 \) and \( \text{int} / t_2 \)
  - Suppose \( \tau_b = \text{int} \rightarrow t_2 \), \( t_3 = \text{bool} \rightarrow \text{bool} \)
  - Can we unify \( \tau_a \) and \( \tau_b \)? No.

Simple Type Substitution (essential to define Unification)

- Language of types
  - \( \tau ::= b \quad / \quad \text{primitive type, e.g., int, bool} \)
  - \( t \quad / \quad \text{type variable} \)
  - \( \tau \rightarrow \tau \quad / \quad \text{function type} \)
- A substitution is a map
  - \( S: \text{Type Variable} \rightarrow \text{Type} \)
  - \( S = [\tau_1/t_1, \ldots, \tau_n/t_n] \quad / \quad \text{substitute type } \tau_i \text{ for type } t_i \)
- A substitution instance \( \tau' = S \tau \)
  - \( S = [t_0/\text{bool} / t_1] \quad / \quad \text{then} \)
  - \( S(\tau) = S(t_0/\text{bool}) \rightarrow (t_0/\text{bool}) \)

Examples

- Substitutions can be composed
  - \( S_1 = [t_0/\text{bool} / t_1] \)
  - \( S_2 = [\text{int} / t_2] \)
  - \( \tau = t_1 \rightarrow t_2 \)
  - \( S_2 S_1(\tau) = ? \)

Example

- \( t_1 \rightarrow \text{bool} = (\text{int} \rightarrow t_2) \rightarrow t_3 \)
- Yes, if \( \text{int}/ t_2 \) and \( \text{bool}/ t_3 \)

Simple Type Substitution (essential to define unification)

- Substitutions can be composed
  - \( S_1 = [t_0/\text{bool} / t_1] \)
  - \( S_2 = [\text{int} / t_2] \)
  - \( \tau = t_1 \rightarrow t_2 \)
  - \( S_2 S_1(\tau) = S_2 [S_1(t_0/\text{bool})] = (\text{int}/\text{bool}) \rightarrow (\text{int}/\text{bool}) \)

Examples

- Substitutions can be composed
  - \( S_1 = [t_1 / t_2] \)
  - \( S_2 = [t_0 / t_2] \)
  - \( S_3 = [t_0/\text{int} / t_1] \)
  - \( \tau = t_1 \rightarrow t_0 \)
  - \( S_2 S_1(\tau) = ? \)
Some Terminology...

- A substitution \( S_1 \) is less specific (i.e., more general) than substitution \( S_2 \) if \( S_2 = S \ S_1 \) 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 \( S_2 = S \ S_1 \) 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 \( S_2 \) satisfying \( C \)

Unification (essential for type inference!)

- \text{Unify}: tries to unify \( \tau_1 \) and \( \tau_2 \) and returns a principal unifier for \( \tau_1 = \tau_2 \) if unification is successful
  def \text{Unify}(\tau_1,\tau_2) =
  case (\tau_1,\tau_2)
    (\tau_1,\tau_2) = [\tau_1,\tau_2] \text{ provided } t_2 \text{ does not occur in } \tau_1
    (\tau_1,\tau_2) = [\tau_2,\tau_1] \text{ provided } t_1 \text{ does not occur in } \tau_2
    \text{if (eq? b_1 b_2) then } [] \text{ else fail}
    (\tau_1 \rightarrow t_{31}, \tau_2 \rightarrow t_{32}) = \text{let } S_1 = \text{Unify}(\tau_1,\tau_2)
    S_2 = \text{Unify}(S_1(\tau_{31}),S_1(\tau_{32}))
    \text{in } S_2 S_1 \text{// compose substitutions}
  otherwise = \text{fail}

Unify Set of Constraints \( C \)

- \text{UnifySet}: tries to unify \( C \) and returns a principal unifier for \( C \) if unification is successful
  def \text{UnifySet}(C) =
  if \( C \) is Empty Set then []
  else let
    \( C = \{ \tau_1 = t_2 \} \cup C' \)
    \( S = \text{Unify}(\tau_1,\tau_2) \)
    in
    \text{UnifySet}(S(C')) * S
  // Composition of substitutions
Outline

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

Add a New Attribute, Substitution Map

Grammar rule:

| E ::= | x | T_E = Γ_E(x) S_E = [ ] |
| E ::= c | T_E = int S_E = [ ] |
| E ::= λx.E1 | T_E1 = Γ_E(x) S_E1 = S_E1 |
| E ::= E1 E2 | Γ_E1 = Γ_E Γ_E2 = S_E1(Γ_E) |
| S = Unify(S_E1(T_E1), T_E2 T_E1) |
| T_E = S(t_E2) S_E = S S_E2 S_E1 |

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!

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

Steps at 1. Finally:
1. \(\Gamma_1 = [[\text{int}\to\text{int}]\to\text{int}]\)
2. \(\Gamma_1 = [\text{int}\to\text{int}]\) returns \(\Gamma = [\text{int}\to\text{int}]\)
3. \(\text{Var} = x\)
4. \(\text{Var}\) is inferred/checked before the type of function body \(E_2\)

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
  \[
  \frac{
  \Gamma \vdash E_1 : \sigma \quad \Gamma ; x : \tau \vdash E_2 : \tau
  }{
  \Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau
  }
  \]
- In static semantics \(\text{let } x = E_1 \text{ in } E_2\) is not equivalent to \((\lambda x.E_2) E_1\)
- In 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

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

Attribute grammar rule

\[ E ::= \lambda x. E \]
\[ E ::= \text{let } x = E_1 \in E_2 \]
\[ E ::= \text{letrec } \Gamma \]

Algorithm W, Almost There!

def \( W(E) \) = case \( E \) of
  c -> (\( c \), TypeOf\( c \))
  x -> if (\( x \) NOT in \( \text{Dom}(\bar{\Gamma}) \)) then fail
      else let \( T_x = \bar{\Gamma}(x) \);
      in (\( \bar{\Gamma}, T_x \))

\( \lambda x. E \) \( \rightarrow \) let \( \Gamma \backslash x = T \in \Gamma \) \( \rightarrow \)
\( E \in \Gamma \)
\( \text{letrec } \Gamma \vdash E_1 \in E_2 \)

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

Example:

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

Unify\( (\text{plus}, \bar{\Gamma}(x)) = W(\text{plus}(x)) \)

Spring 19 CSCI 4450/6450, A. Milanova 39

The Letrec Construct

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

Ax::= letrec \( x = E_1 \) in \( E_2 \)

Attribute grammar rule

\[ \Gamma = \bar{\Gamma}(x) \]
\[ \Gamma_1 = \bar{\Gamma}(x) \]
\[ \Gamma_2 = \bar{\Gamma}(x) \]
\[ \Gamma_3 = \bar{\Gamma}(x) \]
\[ \Gamma_4 = \bar{\Gamma}(x) \]

Unify\( (\text{plus}, \bar{\Gamma}(x)) = W(\text{plus}(x)) \)

Algorithm W, Almost There!

def \( W(E) \) = case \( E \) of
  c -> (\( c \), TypeOf\( c \))
  x -> if (\( x \) NOT in \( \text{Dom}(\bar{\Gamma}) \)) then fail
      else let \( T_x = \bar{\Gamma}(x) \);
      in (\( \bar{\Gamma}, T_x \))

\( \lambda x. E \) \( \rightarrow \) let \( \Gamma \backslash x = T \in \Gamma \) \( \rightarrow \)
\( E \in \Gamma \)
\( \text{letrec } \Gamma \vdash E_1 \in E_2 \)

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

Example:

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

Unify\( (\text{plus}, \bar{\Gamma}(x)) = W(\text{plus}(x)) \)

Spring 19 CSCI 4450/6450, A. Milanova 40
W is Standard Recursive Descend

\[ W(i, E) = \begin{cases} 
\text{case } E \text{ of} & \\
\text{App } E_1 E_2 \to \text{ let} & \\
& s_1 = W(i, E_1) \\
& \ldots \\
& s_2 = W(i_2, E_2) \\
& \text{ in} \\
& s = g(i, s_1, i_2, s_2) \\
\end{cases} \]

Outline

- Simple type inference
- Equality constraints
- Unification
- Strategy 1: Constraint-based typing
- Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism (next time)
- Hindley Milner type inference. Algorithm W