Announcements

- Submitty page for HW5 up!
- Due Thursday
- HW6 coming up tonight
- Paper presentation guidelines coming up tomorrow
- Papers coming up

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

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

Reading

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

Static Semantics
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.

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)

Type Constraints

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

\[
\begin{align*}
  t_2 & = t_4 \rightarrow t_3 & t_3 & = \text{int} \rightarrow t_3 & t_4 & = \text{int} \rightarrow \text{int} \\
  t_2 & = t_7 \rightarrow t_3 & t_3 & = \text{int} & t_4 & = \text{int} \rightarrow \text{int} \\
  t_4 & = t_6 \rightarrow t_5 & t_5 & = \text{int} \rightarrow \text{int} & t_6 & = \text{int} \rightarrow \text{int} \\
  t_7 & = \text{int} \rightarrow t_3 & t_8 & = \text{int} & t_9 & = \text{int} \\
  t_5 & = \text{int} & t_9 & = \text{int} & t_5 & = \text{int} \\
  (\lambda f. \text{int} \rightarrow \text{int} \ f 5) (\lambda x. \text{int}. x+1) : \text{int} (t_5)
\end{align*}
\]

We Can Infer All Types!

\[
\begin{align*}
  \lambda f. f 5, \lambda x. x+1 : ? \\
  & \text{1. App } \Gamma = [] \quad t_2 = t_3 \rightarrow t_1 \\
  & \text{2. Abs } \Gamma = [] \quad t_3 = t_4 \rightarrow t_5 \\
  & \text{3. App } \Gamma = [f: t_4] \quad t_5 = \text{int} \rightarrow t_5 \\
  & \text{4. Abs } \Gamma = [f: t_4] \quad t_5 = t_6 \rightarrow t_5 \\
  & \text{5. } \Gamma = [x: t_3] \quad t_5 = \text{int} \rightarrow \text{int} \\
\end{align*}
\]

Another Example

- \(\text{twice } f \ x = f (f \ x)\)
- What is the type of \(\text{twice}\)?
  - It is \(t_1 \rightarrow t_1 \rightarrow t_2 (t_1 \text{ is the type of } f (f \ x))\)
  - Based on the syntax tree of \(f (f \ x)\) we have:
    \[
    \begin{align*}
    t_2 & = t_2 \rightarrow t_1 \\
    t_1 & = t_2 \rightarrow t_2
    \end{align*}
    \]
  - Thus, \(t_2 = t_1 = t_2, t_3 = t_4 \rightarrow t_2\) and type of \(\text{twice}\) is \(t_1 \rightarrow t_1 \rightarrow t_2\)
- Note: \(t_5\) is a free type variable! Polymorphism!
Type Constraints from Typing
Rules, as Attribute Grammar

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

Grammar rule:

- Attribute rule:

\[
\begin{align*}
E &::= x \quad C_E = \{ t_E = \Gamma_E(x) \} \\
E &::= c \quad C_E = \{ t_E = \text{int} \} \\
E &::= \lambda x.E_1 \quad \Gamma_{E_1} = \Gamma_E; x:t_E \quad C_E = C_{E_1} \cup \{ t_E = t_{E_1} \} \\
E &::= E_1 E_2 \quad \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \quad C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = t_{E_2} \rightarrow t_E \} \\
E &::= E_1 + E_2 \quad \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \quad C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = \text{int}, t_{E_2} = \text{int}, t_E = \text{int} \}
\end{align*}
\]

Example

- \( \lambda f.\lambda x. f(f x) \)

Standard Way of Writing This...

- Semantic rules over syntax, generate constraints, i.e., attribute grammar!
- E.g., rule for abstraction \( A \)
  \[
  \Gamma \vdash \lambda x.E_1 : t_E \quad \exists \tau_a. ( \Gamma ; x : \tau_a \vdash E_1 : \tau_E )
  \quad \tau_a = \tau \quad \tau = t_2 \rightarrow a
  \]
  This reads: Constraints for abstraction term \( A \) given env. \( \Gamma \) include all constraints generated for term \( E_1 \) given augmented env. \( \Gamma ; x : \tau_a \), and constraint \( t = t_2 \rightarrow a \) for term \( A \) itself. \( \tau_a \) 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
- 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 \( \tau_a = \tau_b \) equiv. to \( t_1 = t_3 \rightarrow t_2 = t_4 \) 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.

Simple Type Substitution

- Language of types
  - \( \tau ::= b \) // primitive type, e.g., \( \text{int} \), \( \text{bool} \)
  - \( t \) // type variable
  - \( \tau \rightarrow \tau \) // function type

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

- A substitution instance \( \tau' = S \tau \)
  - \( S = [ t_0 \rightarrow \text{bool} / t_1 ] \) \( \tau = t_1 \rightarrow t_2 \) then
  - \( S(t_1 \rightarrow t_2) = (t_0 \rightarrow \text{bool}) \rightarrow (t_0 \rightarrow \text{bool}) \)

Simple Type Substitution (essential to define Unification)

- Substitutions can be composed
  - \( S_1 = [ t_0 \rightarrow \text{bool} / t_1 ] \)
  - \( S_2 = [ \text{int} / t_3 ] \)
  - \( \tau = t_1 \rightarrow t_2 \)
  - \( S_2 S_1 (\tau) = S_2(S_1(t_1 \rightarrow t_2)) \)
  - \( S_2(S_1(t_1 \rightarrow t_2)) = (t_0 \rightarrow \text{bool}) \rightarrow (t_0 \rightarrow \text{bool}) \)
  - \( (t_0 \rightarrow \text{bool}) \rightarrow (t_0 \rightarrow \text{bool}) \)

Examples

- Substitutions can be composed
  - \( S_1 = [ t_1 / t_2 ] \)
  - \( S_2 = [ t_3 / t_1 ] \)
  - \( S_3 = [ t_4 \rightarrow \text{int} / t_1 ] \)
  - \( \tau = t_1 \rightarrow t_2 \)
  - \( S_2 S_1 (\tau) = ? \)
  - \( S_2 S_1 (\tau) = ? \)
Some Terminology...

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

**Examples**

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

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.
- def Unify($\tau_1, \tau_2$) =
  
  \[
  \begin{aligned}
  \text{case } (\tau_1, \tau_2) \\
  & (\tau_1, \tau_2) = [\tau_1, \tau_2] \text{ provided } \tau_2 \text{ does not occur in } \tau_1 \\
  & (\tau_1, \tau_2) = [\tau_2, \tau_1] \text{ provided } \tau_1 \text{ does not occur in } \tau_2 \\
  & (b_1, b_2) = \text{if } (\text{eq? } b_1, b_2) \text{ then } \text{fail} \\
  & (\tau_1 \rightarrow \tau_{12}, \tau_{21} \rightarrow \tau_{22}) = \text{let } S_1 = \text{Unify}((\tau_1, \tau_{21})) \\
  & \quad S_2 = \text{Unify}(S_1, (\tau_{12}, \tau_{22})) \\
  & \quad \text{in } S_2 S_1 \text{// compose substitutions} \\
  \text{otherwise } = \text{fail}
  \end{aligned}
  \]

**Examples**

- Unify (int \rightarrow int, t_1 \rightarrow t_2) returns ?
- Unify (int, int \rightarrow t_2) returns ?
- Unify (t_1 = \text{int} \rightarrow t_2) returns ?

**Unify Set of Constraints $C$**

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

**Examples**

- $\{ t_1 = \text{int}, t_2 = t_1 \rightarrow t_1 \}$
- $\{ t_1 \rightarrow t_2 = t_2 \rightarrow t_3, t_3 = t_4 \rightarrow t_5 \}$
- $\{ t_1 = t_2 \rightarrow t_1, t_4 = t_5 \rightarrow t_2 \}$
- $\{ t_2 = t_4 \rightarrow t_1, t_2 = t_4 \rightarrow t_3, t_4 = t_5 \rightarrow t_6, t_1 = \text{int} \rightarrow t_3, t_5 = \text{int} \rightarrow t_6, t_4 = \text{int} \rightarrow t_6 \}$
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

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

Grammar rule:

| E ::= x         | T_E = T_E(x) S_E = [] |
| E ::= c         | T_E = int S_E = []    |
| E ::= λx.E_1   | T_E = T_E1(x) T_E1 S_E = S_E1 |
| E ::= E_1 E_2  | Γ_E1 = Γ_E2 Γ_E2 S_E2 Γ_E2(T_E2 → T_E1) 
  S = Unify(S_E2(T_E1), T_E2) S_E2 S_E1 |

Example: (λf. f 5) (λx. x)

Steps at 1.
1. unify([int-t], [Int-t], [Int-t]) returns S = [int-t, int-t, int-t, int-t, int-t]
2. S_1 = S S_1 = S S_2 = S [Int-t, Int-t, Int-t, Int-t, Int-t]
3. T_1 = S_1 = int

Example: λf.λx. (f (f x))

The Let Construct

- In dynamic semantics, \( \text{let } x = E_1 \text{ in } E_2 \) is equivalent to \( \text{let } x = T_1 \text{ in } E_2 : \tau \)
  - Typing rule
  \[
  \Gamma |- E_1 : \sigma \\
  \Gamma ; x : \sigma |- E_2 : \tau \\
  \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau
  \]
- In static semantics \( \text{let } x = E_1 \text{ in } E_2 \) is not equivalent to \( \text{let } x = T_1 \text{ in } E_2 \)
  - In let, type of the argument \( E_1 \) is inferred/checked before
    type of function body \( E_2 \)
  - let construct enables Hindley Milner style polymorphism!
The Let Construct

- **Typing rule**
  \[ \Gamma \vdash E : \sigma \quad \Gamma \vdash x : \tau \]
  \[ \Gamma \vdash \text{let } x = E \text{ in } E_2 : \tau \]

- **Attribute grammar rule**
  \[ E ::= \text{let } x = E_1 \text{ in } E_2 \]
  \[ \Gamma_1 = \Gamma \quad \Gamma_2 = \Gamma_1 \{ x : \tau \} \]
  \[ \Gamma_3 = \Gamma_2 \{ t_1 : \tau \} \]
  \[ \tau = \text{TypeOf } t_1 \]
  
  Note: Can merge let and letrec in let

The Letrec Construct

- **letrec**
  \[ \text{letrec } x = E_1 \text{ in } E_2 \]
  x can be referenced from within \( E_1 \)

  - Extends calculus with general recursion
    - No need to type \( \text{fix} \) (we can’t) but we can type recursive functions like \( \text{plus} \), \( \text{times} \), etc.
  - Haskell’s \textbf{let} is a \textbf{letrec} actually...

  - \( \text{E.g.,} \)
    \[ \text{letrec } \lambda x.\lambda y. \text{if } (x=0) \text{ then } y \text{ else } ((\text{plus } x-1) \text{ y}+1) \]
    written as
    \[ \text{letrec } x \ y = \text{if } (x=0) \text{ then } y \text{ else } \text{plus } (x-1) \ (y+1) \]

Algorithm W, Almost There!

\[ \text{def } W'(E) = \text{case } E \text{ of} \]
\[ c \rightarrow (\langle \text{c}, \text{TypeOf}(c) \rangle) \text{ if } \text{(x NOT in Dom("\text{c}")) then fail} \]
\[ \text{else let } T = \Gamma(x); \text{ in } (\langle T \rangle, T) \]
\[ \lambda x. E_1 \rightarrow \text{let } (S, T) = W'((x:1), E_1) \text{ in } (S_1:1, T_1) \]
\[ E_1, E_2 \rightarrow \text{let } S = W'(E_1); T = W'(E_2) \]
\[ (S, T) = \text{Unify}(S_1, T; T; S_2, T_2) \text{ in } (S_1:1, T_1; T_2:1) \]
\[ \text{if } S \ S_2 \ S_1 \text{ composes substitutions} \]
\[ \text{let } x = E_1; E_2 \rightarrow \text{let } (S, T) = W'((x:1), E_2) \]
\[ (S_1:1, T_1) = \text{Unify}(S_1, T_1; T_1:1) \text{ in } (S_1:1, T_1:1) \]

Algorithm W, Almost There!

(merges let and letrec)

\[ \text{def } W'(E) = \text{case } E \text{ of} \]
\[ c \rightarrow (\langle \text{c}, \text{TypeOf}(c) \rangle) \text{ if } \text{(x NOT in Dom("\text{c}")) then fail} \]
\[ \text{else let } T = \Gamma(x); \text{ in } (\langle T \rangle, T) \]
\[ \lambda x. E_1 \rightarrow \text{let } (S, T) = W'((x:1), E_1) \text{ in } (S_1:1, T_1) \]
\[ E_1, E_2 \rightarrow \text{let } S = W'(E_1); T = W'(E_2) \]
\[ (S, T) = \text{Unify}(S_1, T; T; S_2, T_2) \text{ in } (S_1:1, T_1; T_2:1) \]
\[ \text{if } S \ S_2 \ S_1 \text{ composes substitutions} \]
\[ \text{let } x = E_1; E_2 \rightarrow \text{let } (S, T) = W'((x:1), E_2) \]
\[ (S_1:1, T_1) = \text{Unify}(S_1, T_1; T_1:1) \text{ in } (S_1:1, T_1:1) \]
W is Standard Recursive Descend

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

Outline

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