Simple Type Inference
Announcements

- We will try to keep calm and carry on…

- HW5 due on Thursday?

- HW6 coming up soon

- Paper presentation guidelines and papers coming up
Announcements

- Synchronous lectures via WebEx during our usual class time. Will record and post
  - Mute your microphones
  - Unmute and interrupt if you have question
  - Please, ask and answer questions
- Synchronous office hours via WebEx

- May change if we experience persistent disruption and technical glitches
Announcements

- Quizzes
  - During class, Submitty gradable due at 5-6pm on the same day (subject to change)
  - Work together, use WebEx chat, instant messaging, etc.
Before…

- 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
    - Stuck states
- 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 (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 Static Semantics

- **Variable**
  \[ \text{Looks up the type of } x \text{ in environment } \Gamma \]
  \[
  \begin{align*}
  &x : \tau \\ &\in \Gamma \\
  &\Gamma |- x : \tau
  \end{align*}
  \]

- **Application**
  \[
  \begin{align*}
  &\Gamma |- E_1 : \sigma \rightarrow \tau \\
  &\Gamma |- E_2 : \sigma \\
  &\Gamma |- (E_1 E_2) : \tau
  \end{align*}
  \]

- **Abstraction**
  \[
  \begin{align*}
  &\text{binding: augments environment } \Gamma \\
  &\text{with binding of } x \text{ to type } \sigma \\
  &\Gamma, x : \sigma |- E_1 : \tau \\
  &\Gamma |- (\lambda x : \sigma. E_1) : \sigma \rightarrow \tau
  \end{align*}
  \]
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  \( x \quad \Gamma = [x: \text{int}, y: \text{bool}] \)
   \( t_3 = \text{int} \)

1 is the top-level term, and 2,3 are subcomponents of the term. We will be deducing types for each of these terms.
Deducing Types

- \((\lambda f: \text{int} \rightarrow \text{int}. \ f \ 5) \ (\lambda x: \text{int}. \ x+1) : ?\)

- \(\lambda z: \text{int}. \ \lambda x: \text{bool}. \ (\lambda y: \text{int}. \ y) \ z : ?\)

- \(\lambda x: \text{int} \rightarrow \text{bool} \rightarrow \text{int} \rightarrow \text{int}. \ \lambda y: \text{int} \rightarrow \text{bool}. \ \lambda z: \text{int}. \ x \ z \ (y \ z) : ?\)
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 derive set of 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
   \[ \Gamma = [] \]
   \[ t_2 = t_4 \rightarrow t_1 \]

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

\[ \Gamma |- E_1 : \text{int} \quad \Gamma |- E_2 : \text{int} \]
\[ \Gamma |- E_1 + E_2 : \text{int} \]
\[ \Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma \]
\[ \Gamma |- (E_1 \, E_2) : \tau \]
We constructed a system of type constraints

Let’s solve the system of constraints

\[ t_2 = t_4 \rightarrow t_1 \quad \text{t}_f = \text{int}\rightarrow\text{t}_3 = t_4 = \text{int}\rightarrow\text{int} \]

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

\[ t_4 = t_x \rightarrow t_5 \quad t_4 = \text{int}\rightarrow\text{int} \]

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

\[ t_5 = \text{int}, \ t_x = \text{int} \]

We inferred all t’s!

\[ t_1 = \text{int} \]

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

\[ t_3 = \text{int} \]

\[ t_4 = \text{int}\rightarrow\text{int} \]

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

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

- `twice f x = f (f x)`
- What is the type of `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:
  
  \[
  \begin{align*}
  t_f &= t_2 \rightarrow t_1 \\
  t_f &= t_x \rightarrow t_2
  \end{align*}
  \]

Thus, `t_x = t_1 = t_2`, `t_f = t_x \rightarrow t_x` and the type of `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: 

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

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

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

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

\[ C \] collects constraints. It is synthesized. Propagates bottom-up the tree.

\[ \Gamma \] is inherited. Propagates top-down the tree.

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

\[ E ::= E_1 E_2 \]

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

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

\[ \lambda f.\lambda x. f (f x) \]
Semantic rules over syntax, generate constraints, i.e., attribute grammar!

E.g., rule for abstraction term $A$

$$\left[ \Gamma \vdash \lambda x. E_1 : t \right] = \exists t_x, a. \left( \left[ \Gamma ; x : t_x \vdash E_1 : a \right] \land t = t_x \rightarrow a \right)$$

This reads: Constraints for abstraction term $A$ given environment $\Gamma$ include all constraints generated for term $E_1$ given augmented environment $\Gamma ; x : t_x$, and constraint $t = t_x \rightarrow a$, for term $A$ itself. $t_x$ 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 $t_1 \rightarrow t_2 = t_3 \rightarrow t_4$ 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
  \( \tau ::= b \quad // \) primitive type, e.g., int, bool
  \( | t \quad // \) type variable
  \( | \tau \rightarrow \tau \quad // \) function type

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

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

Substitutions can be composed

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

$$S_2 ( (t_0 \rightarrow \text{bool}) \rightarrow (t_0 \rightarrow \text{bool}) ) = (\text{int} \rightarrow \text{bool}) \rightarrow (\text{int} \rightarrow \text{bool})$$
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

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

- $\tau = t_1 \rightarrow t_2$
- $S_3 S_2 S_1 (\tau) = \ ?$
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 $S_2$ satisfying $C$. 

Spring 20 CSCI 4450/6450, A Milanova

27
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 $\tau_2$; 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

- Unify (int→int, t₁→t₂) yields ?
- Unify (int, int→t₂) yields ?
- Unify (t₁, int→t₂) yields ?

- Unify (tₓ→t₂, t₂→t₁)
  - Unify(tₓ,t₂) = [tₓ/t₂]
  - Unify([tₓ/t₂] t₂, [tₓ/t₂] t₁) = Unify(tₓ, t₁) = [tₓ/t₁]
  - Unify (tₓ→t₂, t₂→t₁) returns [tₓ/t₁] [tₓ/t₂]
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 \} \cup C'
        S = Unify(\tau_1, \tau_2)
    in
        UnifySet(S(C')) ° S
    // Composition of substitutions
```
Examples

- \{ t_1 = \text{int}, \ t_2 = t_1 \to t_1 \} \\
- \{ t_1 \to t_2 = t_2 \to t_3, \ t_3 = t_4 \to t_5 \} \\
- \{ t_f = t_2 \to t_1, \ t_f = t_x \to t_2 \} \\
- \{ t_2 = t_4 \to t_1, \ t_2 = t_f \to t_3, \ t_4 = t_x \to t_5, \ t_f = \text{int} \to t_3, \ t_5 = \text{int}, \ t_x = \text{int} \}
Summary of Strategy 1

- First, collect a set of constraints $C$ via traversal of lambda term
  - Essentially, yet another interpreter for the lambda calculus

- Second, solve set of constraints $C$
  - Unification, and
  - Substitution
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
Grammar rule:  

\[
E ::= x \\
E ::= c \\
E ::= \lambda x.E_1 \\
E ::= E_1 E_2
\]

Attribute rule:  

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

$T_E$ 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

\[\Gamma_2 = []\]
\[T_2 = (\text{int}\rightarrow\text{t}_3)\rightarrow\text{t}_3\]
\[S_2 = [\text{int}\rightarrow\text{t}_3/t_f]\]

3. App

\[\Gamma_3 = [f:\text{t}_f]\]
\[T_3 = \text{t}_3\]
\[S_3 = [\text{int}\rightarrow\text{t}_3/t_f]\]

4. Abs

\[\Gamma_4 = S_2(\Gamma_1) = []\]
\[T_4 = \text{t}_x\rightarrow\text{t}_x\]
\[S_4 = []\]

\[\Gamma = [x:\text{t}_x]\]

Steps at 1, finally:
1. unify( \((\text{int}\rightarrow\text{t}_3)\rightarrow\text{t}_3, (\text{t}_x\rightarrow\text{t}_x)\rightarrow\text{t}_1)\) returns \(S = [\text{int/t}_x, \text{int/t}_3, \text{int/t}_1]\)
2. \(S_1 = S S_4 S_2 = S S_2 = S [\text{int}\rightarrow\text{t}_3/t_f]\)
3. \(T_1 = S(t_1) = \text{int}\)

\[T = \text{t}_x\]
\[S = []\]

from Unify(\(\text{t}_f, \text{int}\rightarrow\text{t}_3\))
Example: \( \lambda f. \lambda x. (f (f x)) \)

- Do as an exercise at home