Simply Typed Lambda Calculus, cont. Simple Type Inference
Announcements

- HW5?
- HW6 is posted
Evaluation of Recursive Function, Revisit

\[ Y = \lambda f. (\lambda x. f (x \times)) (\lambda x. f (x \times)) \]

\[ M = \lambda f. \lambda x. \lambda y. \text{if} \ (\text{iszero } x) \ \text{then} \ (f (x-1) (y+1)) \]

\[ \text{plus} = Y M \]

\[ \text{plus } 2 \ 3 = (Y M) \ 2 \ 3 \]

\[ (\lambda x. M (x \times)) (\lambda x. M (x \times)) \]

\[ M \ 2 \ 3 \rightarrow \]

\[ M ((\lambda x. M (x \times)) (\lambda x. M (x \times))) \]

\[ \text{if} (\text{iszero } 2) \ 3 \ ((Y M) (2-1) (3+1)) \]

\[ \text{if} \ \text{false} \ 3 \ ((Y M) (2-1) (3+1)) \]

\[ (Y M) (2-1) (3+1) \quad \rightarrow \quad \text{Succ } 5 \ 0 \]
Evaluation of Recursive Function, Revisit

\[ \text{interpret} (\text{plus}) = \lambda x. \lambda y. \text{if} (\text{iszero} \ x) y \ (\text{plus} \ (x-1) \ (y+1)) \]

\[ \frac{E_1 E_2}{\text{WHNF}} \]

\( (\lambda x. \lambda y. \ \text{if} (\text{iszero} \ x) y \ (\text{plus} \ (x-1) \ (y+1))) \ 2 \ E_2 \rightarrow \)

\( (\lambda y. \ \text{if} (\text{iszero} \ 2) y \ (\text{plus} \ (2-1) \ (y+1))) \ 3 \rightarrow \)

\( \text{if} (\text{iszero} \ 2) \ 3 \ (\text{plus} \ (2-1) \ (3+1)) \rightarrow \)

\( (\text{false} \ 3 \ (\text{plus} \ (-) \ -)) \rightarrow \)

\( \text{plus} \ (2-1) \ (3+1) \rightarrow 5 \)
The simply typed lambda calculus
  - Syntax
  - Static semantics
  - Dynamic semantics
    - Stuck states
  - Type safety = progress + preservation

Introduction to simple type inference
Putting It All Together, Formally

- Simply typed lambda calculus ($\text{System F}_1$)
  - Syntax
  - The type system: type expressions, environment, and type judgments
- The dynamic semantics
  - Stuck states
- Progress and preservation theorem
**Type Expressions**

- **Introducing type expressions**
  \[ \tau ::= b \mid \tau \to \tau \]

  A type is a basic type \( b \) (we will only consider \( \text{int} \), for simplicity), or a function type

- **Examples**
  \[
  \text{int} \\
  \text{int} \to (\text{int} \to \text{int}) \quad // \quad \text{is right-associative}, \text{ thus can write just } \text{int} \to \text{int}
  \]

- **Syntax of simply typed lambda calculus:**
  \[
  E ::= x \mid (\lambda x : \tau. \ E_1) \mid (\ E_1 \ E_2 ) \mid c
  \]
A term in the simply typed lambda calculus is
- Type correct i.e., well-typed, or
- Type incorrect

The rules that judge type correctness are given in the form of type judgments in an environment

Environment $\Gamma \vdash E : \tau$ (\(\vdash\) is the turnstile)

Read: environment $\Gamma$ entails that $E$ has type $\tau$

Type judgment

$\Gamma \vdash E_1 : \sigma \rightarrow \tau$ \quad \Gamma \vdash E_2 : \sigma$

$\Gamma \vdash (E_1 E_2) : \tau$
Semantics

- **(Variable)**: Looks up the type of $x$ in environment $\Gamma$.
- **(Application)**: $\Gamma \vdash (E_1 E_2) : \tau$
- **(Abstraction)**: Binding augments environment $\Gamma$ with binding of $x$ to type $\sigma$.

Symbols:
- $\in$ (membership)
- $\vdash$ (derivation)
- $\rightarrow$ (function type)
- $\mathfrak{V}$ (truth value)
- $\mathfrak{F}$ (false value)
- $\forall$ (for all)
- $\exists$ (there exists)
- $\mathfrak{N}$ (natural numbers)
- $\mathfrak{M}$ (integers)
- $\mathfrak{B}$ (booleans)

Examples:
- $\text{nil} \vdash x : \text{int}$
- $\text{nil} \not\vdash \lambda x : \text{int}. x$
Examples

Deduce the type for

\( \lambda x : \text{int} . \lambda y : \text{bool} . x \) in the \textbf{nil} environment

\[ \text{int} \rightarrow \text{bool} \rightarrow \text{int} \]
Extensions (to Static Semantics)

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

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

(Comparison)

\[ \Gamma \vdash \text{true} : \text{bool} \]

\[ \Gamma \vdash E_1 = E_2 : \text{bool} \]

\[ \Gamma \vdash b : \text{bool} \quad \Gamma \vdash E_1 : \tau \quad \Gamma \vdash E_2 : \tau \]

\[ \Gamma \vdash \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau \]
Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Validity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Nil} \vdash \lambda x: \text{int}. \lambda y: \text{bool}. x+y : \text{int} \rightarrow \text{bool} \rightarrow \text{int}$</td>
<td>❌</td>
</tr>
<tr>
<td>$\text{Nil} \vdash \lambda x: \text{bool}. \lambda y: \text{int}. \text{if } x \text{ then } y \text{ else } y+1 : \text{bool} \rightarrow \text{int} \rightarrow \text{int}$</td>
<td>✓</td>
</tr>
</tbody>
</table>
Can we deduce the type of this term?

\(\lambda f. \lambda x. \text{if } x=1 \text{ then } x \text{ else } (f (f (x-1))) : ?\)

\[
\begin{align*}
\Gamma |\vdash & E_1 : \text{int} & \Gamma |\vdash & E_2 : \text{int} \\
\hline
\Gamma |\vdash & E_1 = E_2 : \text{bool} \\
\hline
\Gamma |\vdash & E_1 : \text{int} & \Gamma |\vdash & E_2 : \text{int} \\
\hline
\Gamma |\vdash & E_1 + E_2 : \text{int} \\
\hline
\Gamma |\vdash & b : \text{bool} & \Gamma |\vdash & E_1 : \tau & \Gamma |\vdash & E_2 : \tau \\
\hline
\Gamma |\vdash & \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau
\end{align*}
\]
Examples

- How about this
  \((\lambda x. x (\lambda y. y) (x \, 1)) \, (\lambda z. z) : ?\)

  \[ x : \text{int} \rightarrow ? \]
  \[ x : \text{int} \rightarrow ? \]

- \(x\) cannot have two “different” types
  - \((x \, 1)\) demands \text{int} \rightarrow ?
  - \((x \, (\lambda y. y))\) demands \((\tau \rightarrow \tau) \rightarrow ?\)

- Program does not reach a “stuck state” but is nevertheless rejected. A sound type system typically rejects some correct programs
Putting It All Together, Formally

- Simply typed lambda calculus (**System F₁**)
  - Syntax of the simply typed lambda calculus
  - The type system: type expressions, environment, and type judgments

- The dynamic semantics
  - Stuck states

- Progress and preservation theorem
Core Dynamic Semantics

- Syntax: \( E ::= c \ | \ x \ | \ ( \lambda x. \ E_1 ) \ | \ ( \ E_1 \ E_2 ) \)
  - \( c \) is integer constant
- Values: \( V ::= \lambda x. \ E_1 \ | \ c \)
- A “call by value” semantics:

- Stuck states: terms that are syntactically valid but aren’t values and cannot be reduced
  - E.g., \( x, c \ c, \ c (\lambda x. \ E_1) \), etc.
Extensions

\[ E_1 \rightarrow E_4 \]

\[ \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \rightarrow \text{if } E_4 \text{ then } E_2 \text{ else } E_3 \]

\[ \text{if True then } E_2 \text{ else } E_3 \rightarrow E_2 \]

\[ \text{if False then } E_2 \text{ else } E_3 \rightarrow E_3 \]
Core Typing Rules (Again…)

Type expressions:
\[ \tau ::= \text{int} | \tau \rightarrow \tau \]

Environment:
\[ \Gamma ::= \text{Nil} | \Gamma, x : \tau \]

\[ \Gamma |- c : \text{int} \]

\[ \text{x : } \tau \in \Gamma \]

\[ \Gamma |- x : \tau \]

\[ \Gamma, x : \sigma |- E_1 : \tau \]

\[ \Gamma |- (\lambda x. E_1) : \sigma \rightarrow \tau \]

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

\[ \Gamma |- (E_1 E_2) : \tau \]
Soundness Theorem, Formally

- **Definition**: $E$ can get stuck if there exist an $E'$ such that $E \xrightarrow{*} E'$ and $E'$ is stuck

  \[ E \Rightarrow E_1 \Rightarrow E_2 \Rightarrow \cdots \Rightarrow E' \]

- **Theorem (Soundness)**: If $\text{Nil} \vdash E : \tau$ and $E \xrightarrow{n} E'$, then $E'$ is a value, or $E' \xrightarrow{} E''$

- **Lemma (Preservation)**: If $\text{Nil} \vdash E : \tau$ and $E \Rightarrow E'$ then $\text{Nil} \vdash E' : \tau$

- **Lemma (Progress)**: If $\text{Nil} \vdash E : \tau$ then $E$ is a value or there exist $E'$ such that $E \Rightarrow E'$

Program Analysis CSCI 4450/6450, A Milanova
Progress, Proof Sketch

- Induction on the structure of the term $E$ (as usual). Assuming Progress holds for component terms, prove that it holds for composite term $E$

1. **Var**: $\text{Nil} \vdash x : \tau$ --- impossible because $\text{Nil} \vdash E : \tau$

2. **Constant**: $\text{Nil} \vdash c : \text{int}$ --- $E$ is a value

3. **Abs**: $\text{Nil} \vdash (\lambda x. E_1) : \tau$ --- again, $E$ is a value

4. **App**: $\text{Nil} \vdash (E_1 E_2) : \tau$

We have $\text{Nil} \vdash E_1 : \sigma \rightarrow \tau$ and $\text{Nil} \vdash E_2 : \sigma$ or otherwise $E$ wouldn’t have been well-typed. Continued…
4. App: \textbf{Nil} \vdash E_1 \ E_2 : \tau. We have \textbf{Nil} \vdash E_1 : \sigma \rightarrow \tau \text{ and } \textbf{Nil} \vdash E_2 : \sigma \text{ or otherwise } E \text{ wouldn’t have been well-typed}

1. If \( E_1 \) is not a value, then \( E_1 \rightarrow E_3 \). (Progress holds for \( E_1 \) by inductive hypothesis.) Thus, \( E_1 \ E_2 \rightarrow E_3 \ E_2 \)

2. If \( E_1 \) is a value but \( E_2 \) is not a value, then \( E_2 \rightarrow E_3 \). (Again, Progress holds for \( E_2 \) by the inductive hypothesis.) Thus, \( V \ E_2 \rightarrow V \ E_3 \)

3. Finally, if \( E_1 \) and \( E_2 \) are both values, then \( E_1 \) must be \( \lambda x. E_3 \) (this is actually by a lemma, the Canonical Forms lemma). Thus, evaluation rule \((\lambda x. E_3) \ V \rightarrow E_3[V/x]\) applies. Done!
Preservation, Proof Sketch

- Similarly, by induction on the structure of term $E$. Assuming Preservation holds for component terms, prove that it holds for term $E$.

1. **Var**: $x$ --- ...

2. **Constant**: $\text{Nil} \vdash c : \text{int}$ --- ...

3. **Abs**: $\text{Nil} \vdash (\lambda x. E_1) : \tau$ --- ...

4. **App**: $\text{Nil} \vdash (E_1 E_2) : \tau$ --- ... Trickier because need to properly account for substitution!
Soundness

Soundness, worth restating

For every state (i.e., term $E$) the program reaches, $E$ is well-typed (by Preservation)

Since $E$ is well-typed, then it is either a value, or it can be further reduced (by Progress)

Therefore, no state the program ever reaches is a “stuck” state
Extensions

- Dynamic semantics and static semantics for
  - Arithmetic,
  - Booleans,
  - Records,
  - Unions,
  - Recursive types,
  - Imperative features,
  - etc., etc.

Safety = Progress + Preservation
Outline

- The simply typed lambda calculus
  - Syntax
  - Static semantics
    - Stuck states
  - Dynamic semantics
  - Type safety = progress + preservation

Introduction to simple type inference
Deducing Types

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

1. Abs

\[ \Gamma = [] \]

\[ t_1 = \text{int} \rightarrow \text{bool} \rightarrow \text{int} \]

\[ \Gamma = [x: \text{int}] \]

\[ t_2 = \text{bool} \rightarrow \text{int} \]

\[ \Gamma = [x: \text{int}, y: \text{bool}] \]

\[ t_3 = \text{int} \]

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 derive type constraints
  - Solve type constraints (offline)
  - Aka constraint-based typing (e.g., Pierce)
We Can Infer All Types!

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

1. **App**
   \(\Gamma = []\)
   \(t_2 = t_4 \rightarrow t_1\)

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

2. **Abs**

\(\lambda f: t_f\)

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

\(\text{Const } 5\)

3. **App**

\(t_f = \text{int} \rightarrow t_3\)

\(\lambda x: t_x\)

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

4. **Abs**

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

\(\text{Var } x\)

\(\text{Const } 1\)

5. **Abs**

\(t_5 = \text{int}\)

\(t_4 = t_x \rightarrow t_5\)

\(\Gamma = \{\}\)

\(\Gamma \vdash E_1 : \text{int}\)

\(\Gamma \vdash E_2 : \text{int}\)

\(\Gamma \vdash E_1 + E_2 : \text{int}\)

\(\Gamma \vdash E_1 : \sigma \rightarrow \tau\)

\(\Gamma \vdash E_2 : \sigma\)

\(\Gamma \vdash (E_1 \ E_2) : \tau\)
Type Constraints

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

\[
\begin{align*}
    t_2 & = t_4 \rightarrow t_1 \\
    t_2 & = t_f \rightarrow t_3 \\
    t_4 & = t_x \rightarrow t_5 \\
    t_f & = \text{int} \rightarrow t_3 \\
    t_5 & = \text{int}, t_x = \text{int}
\end{align*}
\]

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

We inferred all \( t \)'s!
\[
\begin{align*}
    t_1 & = \text{int} \\
    t_2 & = (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \\
    t_3 & = \text{int} \\
    t_4 & = \text{int} \rightarrow \text{int} \\
    t_f & = \text{int} \rightarrow \text{int}
\end{align*}
\]
Another Example

- \( \text{twice } f \ x = f \ (f \ x) \)
- What is the type of `twice`?
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:
  
  \[ t_f = t_2 \rightarrow t_1 \]  
  \[ t_f = t_x \rightarrow t_2 \]  

Thus, `t_x = t_1 = t_2`, `t_f = t_x \rightarrow t_x` and type of `twice` is `(t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x`.

Note: `t_x` is a free type variable! Polymorphism!