Simply Typed Lambda Calculus, cont. Simple Type Inference
Announcements

- HW5?
- HW6 is posted
Outline

- 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 (System $F_1$)
  - 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
Type Expressions

- Introducing type expressions
  - $\tau ::= b \mid \tau \to \tau$
  - A type is a basic type $b$ (we will only consider `int`, for simplicity), or a function type

- Examples
  - `int`
  - `int \to (\text{int} \to \text{int})` is right-associative, thus can write just `int \to \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 \( \Gamma |- E : \tau \) (\( |- \) is the turnstile)
- Read: environment \( \Gamma \) entails that \( E \) has type \( \tau \)

Type judgment
\[
\begin{align*}
\Gamma |- E_1 : \sigma \rightarrow \tau & \quad \Gamma |- E_2 : \sigma \\
\hline
\Gamma |- (E_1 \ E_2) : \tau
\end{align*}
\]
Semantics

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

- **Application**
  \[ \Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma \quad \Gamma |- (E_1 \ E_2) : \tau \]

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

- Binding: augments environment \( \Gamma \) with binding of \( x \) to type \( \sigma \)
Examples

Deduce the type for

\[ \lambda x: \text{int}. \lambda y: \text{bool}. \ x \text{ in the nil environment} \]
Extensions

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

\[ \Gamma |- E_1 : \text{int} \quad \Gamma |- E_2 : \text{int} \quad (\text{Comparison}) \]

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

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

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

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

Is this a valid type?

```
Nil |- \(\lambda x:\text{int}.\lambda y:\text{bool}. x+y : \text{int} \rightarrow \text{bool} \rightarrow \text{int}
```

- No. It gets rightfully rejected. Term reaches a “stuck state” as it applies \(+\) on a value of the wrong type (\(y\) is \text{bool}, \(+\) is defined on \text{ints})

Is this a valid type?

```
Nil |- \(\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}
```

Examples

Is this a valid type?

\(
\text{Nil} \vdash \lambda x: \text{bool.} \lambda y: \text{int. if } x \text{ then } y \text{ else } y+1 : \text{bool } \rightarrow \text{int } \rightarrow \text{int}
\)
Examples

Can we deduce the type of this term?

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

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

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

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

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

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

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

- How about this

\[(\lambda x. \ x \ (\lambda y. \ y) \ (x \ 1)) \ (\lambda z. \ z) \ : \ ?\]

- \(x\) 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 F**₁)
  - 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: $E ::= c \mid x \mid (\lambda x. E_1) \mid (E_1 E_2)$
  - $c$ is integer constant
- Values: $V ::= \lambda x. E_1 \mid c$
- A “call by value” semantics:
  
  $$(\lambda x. E) V \rightarrow E[V/x] \quad E_1 \rightarrow E_2 \quad E_1 E_3 \rightarrow E_2 E_3 \quad V E_1 \rightarrow V E_2$$

- 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.
Core Typing Rules (Again…)

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

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

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

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

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

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

\[ \Gamma |- (E_1 \ E_2) : \tau \]

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

Environment:
\[ \Gamma ::= \text{Nil} | \Gamma, \mathsf{x} : \tau \]
Soundness Theorem, Formally

Definition: $E$ can get stuck if there exist an $E'$ such that $E \Rightarrow^* E'$ and $E'$ is stuck.

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

- 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 $E \Rightarrow E'$
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…
Progress, Proof Sketch

4. App: \textbf{Nil} \vdash E_1 \; E_2 : \tau. We have \textbf{Nil} \vdash E_1 : \sigma \rightarrow \tau 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: $\text{Nil} \mid- c : \text{int}$ --- ...
3. Abs: $\text{Nil} \mid- (\lambda x. E_1) : \tau$ --- ...
4. App: $\text{Nil} \mid- (E_1 E_2) : \tau$ --- … 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
  - Dynamic semantics
    - Stuck states
  - Type safety = progress + preservation

- Introduction to simple type inference
Deducing Types

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

1. Abs \( \Gamma = [] \)
   \( t_1 = \text{int} \rightarrow \text{bool} \rightarrow \text{int} \)

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.
Deducing Types

\((\lambda f:\text{int} \to \text{int}. \ f \ 5) \ (\lambda x:\text{int}. \ x+1) : ?\)
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
   \[\Gamma = []\]
   \[t_2 = t_4 \rightarrow t_1\]

2. Abs
   \[\Gamma = []\]
   \[t_2 = t_f \rightarrow t_3\]

\(\lambda f: t_f\)

3. App
   \[\Gamma = [f:t_f]\]
   \[t_f = \text{int} \rightarrow t_3\]

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

Var \(f\)

Const 5

\(\lambda x: t_x\)

4. Abs
   \[\Gamma = []\]
   \[t_4 = t_x \rightarrow t_5\]

5. +
   \[\Gamma = [x:t_x]\]
   \[t_5 = \text{int}\]
   \[t_x = \text{int}\]

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

Var \(x\)

Const 1

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

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

\[\Gamma \vdash E_1 : \sigma \rightarrow \tau \quad \Gamma \vdash E_2 : \sigma\]

\[\Gamma \vdash (E_1, E_2) : \tau\]
Type Constraints

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

\[ t_2 = t_4 \rightarrow t_1 \]  \[ t_f = \text{int} \rightarrow t_3 = t_4 = \text{int} \rightarrow \text{int} \]

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

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

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

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

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

- `twice f x = f (f x)`
- What is the type of `twice`?
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

  type of `twice` is `(t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x`

Note: `t_x` is a free type variable! **Polymorphism!**
Syntax: \( E ::= x \mid c \mid \lambda x. E \mid E_1 E_2 \mid E_1 + E_2 \)

Grammar rule:  

Attribute rule:  

\( E ::= x \)  
\( C_E = \{ t_E = \Gamma_E(x) \} \)

\( E ::= c \)  
\( C_E = \{ t_E = \text{int} \} \)

\( E ::= \lambda x. E_1 \)  
\( \Gamma_{E_1} = \Gamma_E x : t_x \)
\( C_E = C_{E_1} \cup \{ t_E = t_x \rightarrow t_{E_1} \} \)

\( E ::= E_1 E_2 \)  
\( \Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = \Gamma_E \)
\( C_E = C_{E_1} \cup C_{E_2} \cup \{ t_{E_1} = t_{E_2} \rightarrow t_E \} \)

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

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

\[ E ::= E_1 E_2 \]

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

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

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

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

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

\( C \) collects constraints. It is synthesized. Propagates bottom-up the tree.

\( \Gamma \) is inherited. Propagates top-down the tree.
Example

\[ \lambda f. \lambda x. f (f x) \]
Next Week

- Simple type inference
  - Equality constraints
  - Unification

- Polymorphic types
- Hindley-Milner type inference
  - Algorithm W