Simply Typed Lambda Calculus, cont. Simple Type Inference
Announcements

- HW5?

- Will post HW6 next time

- I am still grading HW4

Expr

Var Name : x

Lambda Name Expr : \( \lambda x. E \)

App Expr Expr : \( E_1, E_2 \)
Type Unsafe C and C++

C:

```c
struct str {
    float f;
    short s;
} s, *s;

int *i = &s;
```

C unions!

```c
union uni {
    float f; // 4 bytes
    short i; // 2 bytes
} u;

u.f = 4.222;
```

C++

```cpp
A* foo1() {
    B* foo2(int);
    A* a = new A();
    B* b = (B*)a;
    b->foo2(100);
}
```

```
FORBIDDEN
```
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 F₁**)
  - Syntax
  - The type system: type expressions, environment, and type judgments
  - The dynamic semantics
    - Stuck states
  - Progress and preservation theorem
Type Expressions

- Syntax of simply typed lambda calculus:
  
  \[
  E ::= x \mid (\lambda x : \tau. E_1) \mid (E_1 E_2) \mid c
  \]

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

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

- Examples
  
  \texttt{int, int\rightarrow int, int\rightarrow int\rightarrow int, (int\rightarrow int)\rightarrow int}

  \texttt{int\rightarrow (int\rightarrow int) // \rightarrow is right-associative, thus can write just \texttt{int\rightarrow int \rightarrow int}}
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

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

\[ \Gamma \vdash (E_1 \ E_2) : \tau \]

Premises

Conclusion
Semantics

\[
\begin{align*}
\Gamma, x : \sigma & \vdash E_1 : \tau \\
\Gamma, x : \sigma & \vdash (\lambda x : \sigma. E_1) : \sigma \rightarrow \tau \\
\Gamma & \vdash (E_1 E_2) : \tau \\
\Gamma & \vdash (\lambda x : \sigma. E_1) : \sigma \rightarrow \tau \\
\Gamma & \vdash x : \tau \in \Gamma \\
\end{align*}
\]

(Variable)

(Computation)

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

(Variable)

(Computation)
Examples

- Deduce the type for

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

\[
\begin{align*}
\mathcal{E} & : \text{nil} \\
\{ x : \text{int } \}
\vdash \lambda y : \text{bool}. \ x
\end{align*}
\]

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

\[
\mathcal{E}
\vdash \lambda y : \text{bool}. \ x
\]

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

\[
\mathcal{E}
\vdash \text{int} \to (\text{bool} \to \text{int})
\]
Examples

- Deduce the type for \( \lambda x: \text{int}. \lambda y: \text{bool}. \ x \) in the nil environment
Extensions (of Language and Static Semantics)

\[ \Gamma \vdash c : \text{int} \]

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

\[ \Gamma \vdash E_1 + E_2 : \text{int} \quad \text{(Arithmetic)} \]

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

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

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

- Is this a valid type?

  \( \text{Nil} \vdash \lambda x: \text{int}. \lambda y: \text{bool}. \; x+y : \text{int} \rightarrow \text{bool} \rightarrow \text{int} \)  

  \text{TYPE INCORRECT}

- Is this a valid type?

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

  \( \text{✓} \)
Examples

Can we deduce the type of this term?

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

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

- How about this
  
  \((\lambda x. x (\lambda y. y) (x \ 1)) \ (\lambda z. z) : ?\)
  
  \[\rightarrow (\lambda z. z ) (\lambda y. y) ( (\lambda z. z) \ 1) \rightarrow\]
  
  \[(\lambda y. y ) ((\lambda z. z) \ 1) \rightarrow (\lambda y. y) \ 1 \rightarrow \ 1\]

- \(x\) 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:

$$
\begin{align*}
(\lambda x. E) V & \rightarrow E[V/x] \\
E_1 \rightarrow E_2 & \\
E_1 E_3 & \rightarrow E_2 E_3 \\
V E_1 & \rightarrow V E_2
\end{align*}
$$

- Stuck states: terms that are syntactically valid but aren’t values and cannot be reduced
  - E.g., $x, x ((\lambda x. x) 1), c c, c (\lambda x. 1),$ etc.
Extensions
Core Typing Rules (Again…)

\[
\begin{align*}
\Gamma |- c : \text{int} \\
\chi : \tau \in \Gamma \\
\Gamma |- \chi : \tau \\
\Gamma, \chi : \sigma |- E_1 : \tau \\
\Gamma |- (\lambda \chi. E_1) : \sigma \rightarrow \tau \\
\Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma \\
\Gamma |- (E_1 \ E_2) : \tau
\end{align*}
\]

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

Environment:
\[
\Gamma ::= \text{Nil} \mid \Gamma, \chi : \tau
\]
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 $E$
Progress, Proof Sketch

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

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: $\text{Nil} |- c : \text{int} --- …$
3. Abs: $\text{Nil} |- (\lambda x. E_1) : \tau --- …$
4. App: $\text{Nil} |- (E_1 E_2) : \tau --- …$ 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

- Next time: Simple type inference