Lecture 25
Type Inference
Review: Polymorphic $\lambda$-Calculus

Syntax

\[
e ::= n \mid x \mid \lambda x: \tau. e \mid e_1 e_2 \mid \Lambda X. e \mid e [\tau]
\]

\[
v ::= n \mid \lambda x: \tau. e \mid \Lambda X. e
\]

Dynamic Semantics

\[
E ::= [\cdot] \mid E e \mid v E \mid E [\tau]
\]

\[
e \rightarrow e'
\]

\[
E[e] \rightarrow E[e'] \quad (\lambda x: \tau. e) v \rightarrow e\{v/x\} \quad (\Lambda X. e) [\tau] \rightarrow e\{\tau/X\}
\]
Review: Polymorphic $\lambda$-Calculus

\[ \Delta, \Gamma \vdash n : \text{int} \]

\[ \Delta, \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ ok} \]

\[ \Delta, \Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau' \]

\[ \Delta \cup \{X\}, \Gamma \vdash e : \tau \]

\[ \Delta, \Gamma \vdash \forall X. e : \forall X. \tau \]

\[ \Gamma(x) = \tau \]

\[ \Delta, \Gamma \vdash x : \tau \]

\[ \Delta, \Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Delta, \Gamma \vdash e_2 : \tau \]

\[ \Delta, \Gamma \vdash e_1 e_2 : \tau' \]

\[ \Delta, \Gamma \vdash e : \forall X. \tau' \quad \Delta \vdash \tau \text{ ok} \]

\[ \Delta, \Gamma \vdash e [\tau] : \tau'\{\tau/X\} \]
Polymorphism let us write a doubling function that works for *any* type of function:

\[
\text{double} \triangleq \forall X. \lambda f : X \to X. \lambda x : X. f(f(x)).
\]

The type of this expression is:

\[
\forall X. (X \to X) \to X \to X
\]

You can use the polymorphic function by providing a type:

\[
\text{double [int]} (\lambda n : \text{int}. n + 1) 7
\]
In languages like OCaml, programmers don’t have to annotate their programs with $\forall X. \tau$ or $e [\tau]$. 
Type Inference

In languages like OCaml, programmers don’t have to annotate their programs with $\forall X. \tau$ or $e[\tau]$.

For example, we can write:

```ocaml
let double f x = f (f x)
```

and OCaml will figure out that the type is:

```
('a -> 'a) -> 'a -> 'a
```

which is equivalent to the same System F type:

```
\forall A. (A -> A) -> A -> A
```
In languages like OCaml, programmers don’t have to annotate their programs with $\forall X. \tau$ or $e[\tau]$.

We can also write

```ocaml
double (fun x -> x+1) 7
```

and OCaml will infer that the polymorphic function `double` is instantiated at the type `int`. 
However, polymorphism in OCaml and other MLs, has some restrictions to ensure that type inference remains \textit{decidable}.
ML Polymorphism

However, polymorphism in OCaml and other MLs, has some restrictions to ensure that type inference remains *decidable*.

These restrictions, called *prenex polymorphism*, stipulate that $\forall$s may only appear in the “outermost” position.
ML Polymorphism

However, polymorphism in OCaml and other MLs, has some restrictions to ensure that type inference remains *decidable*.

These restrictions, called *prenex polymorphism*, stipulate that $\forall$s may only appear in the “outermost” position.

**Examples**

- Prenex: $\forall \alpha. \alpha \rightarrow \alpha$
ML Polymorphism

However, polymorphism in OCaml and other MLs, has some restrictions to ensure that type inference remains *decidable*. These restrictions, called *prenex polymorphism*, stipulate that ∀s may only appear in the “outermost” position.

**Examples**

- Prenex: ∀α. α → α
- Not prenex: (∀α. α → α) → int
ML Polymorphism

However, polymorphism in OCaml and other MLs, has some restrictions to ensure that type inference remains *decidable*.

These restrictions, called *prenex polymorphism*, stipulate that $\forall$s may only appear in the “outermost” position.

**Examples**

- **Prenex:** $\forall \alpha. \alpha \to \alpha$
- **Not prenex:** $(\forall \alpha. \alpha \to \alpha) \to \text{int}$

These restrictions have the following practical ramifications:

- Can’t instantiate type variables with polymorphic types
- Can’t put a polymorphic type on the left of an arrow
These restrictions mean that certain terms that are typeable in System F are not typeable in ML!
These restrictions mean that certain terms that are typeable in System F are not typeable in ML!

```
# fun x -> x x;;
Error: This expression has type 'a -> 'b
but an expression was expected of type 'a
The type variable 'a occurs inside 'a -> 'b
```
Type Inference

Type inference may be undecidable for the polymorphic \( \lambda \)-calculus and OCaml, but it is possible for the simply-typed \( \lambda \)-calculus!
Type Inference

Type inference may be undecidable for the polymorphic $\lambda$-calculus and OCaml, but it is possible for the simply-typed $\lambda$-calculus!

Type inference for the STLC means guessing a $\tau$ in every abstraction in an untyped version:

$$\lambda x. e$$

to produce a typed program:

$$\lambda x: \tau. e$$

that we can use in the typing rule for functions.
Example

Here’s an untyped program:

\[ \lambda a. \lambda b. \lambda c. \text{if } a (b + 1) \text{ then } b \text{ else } c \]
Example

Here’s an untyped program:
\[ \lambda a. \lambda b. \lambda c. \text{if } a (b + 1) \text{ then } b \text{ else } c \]

Informal inference:
Example

Here’s an untyped program:
\[
\lambda a. \lambda b. \lambda c. \text{if } a (b + 1) \text{ then } b \text{ else } c
\]

Informal inference:
- \( b \) must be \textbf{int}
Here’s an untyped program:

\[ \lambda a. \lambda b. \lambda c. \text{if } a \ (b + 1) \ \text{then } b \ \text{else } c \]

Informal inference:

- \( b \) must be \texttt{int}
- \( a \) must be some kind of function
Example

Here’s an untyped program:

\[ \lambda a. \lambda b. \lambda c. \text{if } a \ (b + 1) \text{ then } b \text{ else } c \]

Informal inference:

- \( b \) must be \textbf{int}
- \( a \) must be some kind of function
- the argument type of \( a \) must be the same as \( b + 1 \)
Example

Here’s an untyped program:
\[ \lambda a. \lambda b. \lambda c. \text{if } a \ (b + 1) \ \text{then } b \ \text{else } c \]

Informal inference:
- \( b \) must be \texttt{int}
- \( a \) must be some kind of function
- the argument type of \( a \) must be the same as \( b + 1 \)
- the result type of \( a \) must be \texttt{bool}
Here’s an untyped program:
\[ \lambda a. \lambda b. \lambda c. \text{if } a \ (b + 1) \text{ then } b \text{ else } c \]

Informal inference:
- \( b \) must be \textbf{int}
- \( a \) must be some kind of function
- the argument type of \( a \) must be the same as \( b + 1 \)
- the result type of \( a \) must be \textbf{bool}
- the type of \( c \) must be the same as \( b \)
Example

Here’s an untyped program:
\[ \lambda a. \lambda b. \lambda c. \text{if } a \ (b + 1) \text{ then } b \text{ else } c \]

Informal inference:
- \( b \) must be \text{int}
- \( a \) must be some kind of function
- the argument type of \( a \) must be the same as \( b + 1 \)
- the result type of \( a \) must be \text{bool}
- the type of \( c \) must be the same as \( b \)

Putting all these pieces together:
\[ \lambda a:\text{int} \rightarrow \text{bool}. \ \lambda b:\text{int}. \ \lambda c:\text{int}. \ \text{if } a \ (b + 1) \text{ then } b \text{ else } c \]
Let’s automate type inference!
Let’s automate type inference!

We introduce a new judgment:

\[ \Gamma \vdash e : \tau \mid C \]

Given a typing context \( \Gamma \) and an expression \( e \), it generates a set of *constraints*—equations between types.
Let’s automate type inference!

We introduce a new judgment:

$$\Gamma \vdash e : \tau \mid C$$

Given a typing context $\Gamma$ and an expression $e$, it generates a set of constraints—equations between types.

If these constraints are solvable, then $e$ can be well-typed in $\Gamma$.

A solution to a set of constraints is a type substitution $\sigma$ that, for each equation, makes both sides syntactically equal.
Let’s define the type inference judgment for this STLC language:

\[ e ::= x \mid \lambda x : \tau. e \mid e_1 e_2 \mid n \mid e_1 + e_2 \]

\[ \tau ::= \textbf{int} \mid X \mid \tau_1 \rightarrow \tau_2 \]

You can use a type variable \( X \) wherever you want to have a type inferred.
Constraint-Based Typing Judgment

\[
\Gamma(x) = \tau \\
\Rightarrow \Gamma \vdash x : \tau \mid \emptyset \text{ CT-VAR}
\]
Constraint-Based Typing Judgment

\[
\Gamma(x) = \tau \\
\Gamma \vdash x : \tau \mid \emptyset \quad \text{CT-VAR} \\
\Gamma \vdash n : \text{int} \mid \emptyset \quad \text{CT-INT}
\]
Constraint-Based Typing Judgment

\[ \Gamma(x) = \tau \]

\[ \frac{\Gamma \vdash x : \tau | \emptyset}{\text{CT-VAR}} \]

\[ \Gamma \vdash n : \text{int} | \emptyset \]

\[ \frac{\Gamma \vdash e_1 : \tau_1 | C_1 \quad \Gamma \vdash e_2 : \tau_2 | C_2}{\Gamma \vdash e_1 + e_2 : \text{int} | C_1 \cup C_2 \cup \{\tau_1 = \text{int}, \tau_2 = \text{int}\}} \]

\[ \text{CT-ADD} \]
Constraint-Based Typing Judgment

\[ \Gamma(x) = \tau \]
\[ \Gamma \vdash x: \tau \mid \emptyset \quad \text{CT-VAR} \]
\[ \Gamma \vdash n:\text{int} \mid \emptyset \quad \text{CT-INT} \]

\[ \Gamma \vdash e_1: \tau_1 \mid C_1 \quad \Gamma \vdash e_2: \tau_2 \mid C_2 \]
\[ \Gamma \vdash e_1 + e_2: \text{int} \mid C_1 \cup C_2 \cup \{\tau_1 = \text{int}, \tau_2 = \text{int}\} \quad \text{CT-ADD} \]

\[ \Gamma, x: \tau_1 \vdash e: \tau_2 \mid C \]
\[ \Gamma \vdash \lambda x: \tau_1. e: \tau_1 \rightarrow \tau_2 \mid C \quad \text{CT-ABS} \]
Constraint-Based Typing Judgment

\[
\Gamma(x) = \tau \quad \text{CT-VAR} \quad \Gamma \vdash x : \tau \mid \emptyset
\]

\[
\Gamma \vdash n : \text{int} \mid \emptyset \quad \text{CT-INT}
\]

\[
\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad \Gamma \vdash e_1 + e_2 : \text{int} \mid C_1 \cup C_2 \cup \{\tau_1 = \text{int}, \tau_2 = \text{int}\} \quad \text{CT-ADD}
\]

\[
\Gamma, x : \tau_1 \vdash e : \tau_2 \mid C \quad \Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2 \mid C \quad \text{CT-ABS}
\]

\[
\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad X \text{ fresh} \quad C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \rightarrow X\} \quad \Gamma \vdash e_1 e_2 : X \mid C' \quad \text{CT-APP}
\]
A type substitution is a finite map from type variables to types.

Example: The substitution

\[X \mapsto \text{int}, \ Y \mapsto \text{int} \rightarrow \text{int}\]

maps type variable \(X\) to \text{int} and \(Y\) to \text{int} \rightarrow \text{int}. \]
Type Substitution

We can define substitution of type variables formally:

\[
\sigma(X) \equiv \begin{cases} 
\tau & \text{if } X \not\in \text{domain of } \sigma \\
\sigma(X) & \text{if } X \text{ not in the domain of } \sigma
\end{cases}
\]

\[
\sigma(int) \equiv int
\]

\[
\sigma(\tau \rightarrow \tau') \equiv \sigma(\tau) \rightarrow \sigma(\tau')
\]

We don’t need to worry about avoiding variable capture: all type variables are “free.”
We can define substitution of type variables formally:

\[
\sigma(X) \triangleq \begin{cases} 
\tau & \text{if } X \mapsto \tau \in \sigma \\
X & \text{if } X \text{ not in the domain of } \sigma
\end{cases}
\]
Type Substitution

We can define substitution of type variables formally:

\[
\sigma(X) \triangleq \begin{cases} 
\tau & \text{if } X \mapsto \tau \in \sigma \\
X & \text{if } X \text{ not in the domain of } \sigma
\end{cases}
\]

\[
\sigma(\text{int}) \triangleq \text{int}
\]
We can define substitution of type variables formally:

$$\sigma(X) \triangleq \begin{cases} \tau & \text{if } X \mapsto \tau \in \sigma \\ X & \text{if } X \text{ not in the domain of } \sigma \end{cases}$$

$$\sigma(\text{int}) \triangleq \text{int}$$

$$\sigma(\tau \rightarrow \tau') \triangleq \sigma(\tau) \rightarrow \sigma(\tau')$$
Type Substitution

We can define substitution of type variables formally:

\[
\sigma(X) \triangleq \begin{cases} 
\tau & \text{if } X \mapsto \tau \in \sigma \\
X & \text{if } X \text{ not in the domain of } \sigma
\end{cases}
\]

\[
\sigma(\text{int}) \triangleq \text{int}
\]

\[
\sigma(\tau \rightarrow \tau') \triangleq \sigma(\tau) \rightarrow \sigma(\tau')
\]

We don’t need to worry about avoiding variable capture: all type variables are “free.”
Type Substitution

We can define substitution of type variables formally:

\[
\sigma(X) \triangleq \begin{cases} 
\tau & \text{if } X \mapsto \tau \in \sigma \\
X & \text{if } X \text{ not in the domain of } \sigma
\end{cases}
\]

\[
\sigma(\text{int}) \triangleq \text{int}
\]

\[
\sigma(\tau \rightarrow \tau') \triangleq \sigma(\tau) \rightarrow \sigma(\tau')
\]

We don’t need to worry about avoiding variable capture: all type variables are “free.”

Given two substitutions \(\sigma_1\) and \(\sigma_2\), we write \(\sigma_1 \circ \sigma_2\) for their composition: \((\sigma_1 \circ \sigma_2)(\tau) = \sigma_1(\sigma_2(\tau))\).
Unification

Our constraints are of the form $\tau = \tau'$.
Our constraints are of the form $\tau = \tau'$.

We say that a substitution $\sigma$ unifies constraint $\tau = \tau'$ if $\sigma(\tau) = \sigma(\tau')$.

We say that substitution $\sigma$ satisfies (or unifies) set of constraints $C$ if $\sigma$ unifies every constraint in $C$. 
If:
- $\Gamma \vdash e : \tau \mid C$, and
- $\sigma$ satisfies $C$,
then $e$ has type $\tau'$ under $\Gamma$, where $\sigma(\tau) = \tau'$.

If there are no substitutions that satisfy $C$, then $e$ is not typeable.
Unification

If:

- $\Gamma \vdash e : \tau \mid C$, and
- $\sigma$ satisfies $C$,

then $e$ has type $\tau'$ under $\Gamma$, where $\sigma(\tau) = \tau'$.

If there are no substitutions that satisfy $C$, then $e$ is not typeable.

So let’s find a substitution $\sigma$ that unifies a set of constraints $C$!
Unification Algorithm

\[
\text{unify}(\tau; \tau') \equiv \text{(the empty substitution)}
\]

\[
\text{if } \tau = \tau' \text{ then }
\]

\[
\text{unify}(C')
\]

\[
\text{elseif } \tau = X \text{ and } X \text{ not a free variable of } \tau' \text{ then }
\]

\[
\text{unify}(C' f \tau' / X g \mid X \not< \tau')
\]

\[
\text{elseif } \tau' = X \text{ and } X \text{ not a free variable of } \tau \text{ then }
\]

\[
\text{unify}(C' f \tau / X g \mid X \not< \tau)
\]

\[
\text{elseif } \tau = \tau_0 \not< \tau_1 \text{ and } \tau' = \tau'_0 \not< \tau'_1 \text{ then }
\]

\[
\text{unify}(C' [ f \tau_0 = \tau'_0, \tau_1 = \tau'_1 \mid g])
\]

\[
\text{else fail}
\]
Unification Algorithm

\[ unify(\emptyset) \triangleq [] \quad \text{(the empty substitution)} \]
Unification Algorithm

\[ \text{unify}(\emptyset) \triangleq [] \quad \text{(the empty substitution)} \]

\[ \text{unify}\left(\{\tau = \tau'\} \cup C'\right) \triangleq \]

if \(\tau = \tau'\) then

\[ \text{unify}(C') \]
Unification Algorithm

\[ \text{unify}(\emptyset) \triangleq [] \quad (\text{the empty substitution}) \]

\[ \text{unify}(\{ \tau = \tau' \} \cup C') \triangleq \]

if \( \tau = \tau' \) then

\[ \text{unify}(C') \]

else if \( \tau = X \) and \( X \) not a free variable of \( \tau' \) then

\[ \text{unify}(C'\{ \tau' / X \}) \circ [X \mapsto \tau'] \]
Unification Algorithm

\[ unify(\emptyset) \triangleq [] \quad \text{(the empty substitution)} \]

\[ unify(\{\tau = \tau'\} \cup C') \triangleq \]

if \( \tau = \tau' \) then
  \[ unify(C') \]
else if \( \tau = X \) and \( X \) not a free variable of \( \tau' \) then
  \[ unify(C'\{\tau'/X\}) \circ [X \mapsto \tau'] \]
else if \( \tau' = X \) and \( X \) not a free variable of \( \tau \) then
  \[ unify(C'\{\tau/X\}) \circ [X \mapsto \tau] \]
Unification Algorithm

\(\text{unify(∅) } \triangleq [\text{substitution}]\) (the empty substitution)

\(\text{unify}(\{ \tau = \tau' \} \cup C') \triangleq\)
if \(\tau = \tau'\) then
  \(\text{unify}(C')\)
else if \(\tau = X\) and \(X\) not a free variable of \(\tau'\) then
  \(\text{unify}(C'\{ \tau' / X \}) \circ [X \mapsto \tau']\)
else if \(\tau' = X\) and \(X\) not a free variable of \(\tau\) then
  \(\text{unify}(C'\{ \tau / X \}) \circ [X \mapsto \tau]\)
else if \(\tau = \tau_0 \rightarrow \tau_1\) and \(\tau' = \tau'_0 \rightarrow \tau'_1\) then
  \(\text{unify}(C' \cup \{ \tau_0 = \tau'_0, \tau_1 = \tau'_1 \})\)
Unification Algorithm

\(\text{unify}(\emptyset) \triangleq []\) (the empty substitution)

\(\text{unify}(\{\mathit{\tau} = \mathit{\tau}'\} \cup C') \triangleq\)

if \(\mathit{\tau} = \mathit{\tau}'\) then

\(\text{unify}(C')\)

else if \(\mathit{\tau} = X\) and \(X\) not a free variable of \(\mathit{\tau}'\) then

\(\text{unify}(C'\{\mathit{\tau}' / X\}) \circ [X \mapsto \mathit{\tau}']\)

else if \(\mathit{\tau}' = X\) and \(X\) not a free variable of \(\mathit{\tau}\) then

\(\text{unify}(C'\{\mathit{\tau} / X\}) \circ [X \mapsto \mathit{\tau}]\)

else if \(\mathit{\tau} = \mathit{\tau}_0 \rightarrow \mathit{\tau}_1\) and \(\mathit{\tau}' = \mathit{\tau}'_0 \rightarrow \mathit{\tau}'_1\) then

\(\text{unify}(C' \cup \{\mathit{\tau}_0 = \mathit{\tau}'_0, \mathit{\tau}_1 = \mathit{\tau}'_1\})\)

else

\(\text{fail}\)
Unification Properties

The unification algorithm always terminates.
Unification Properties

The unification algorithm always terminates.

The solution, if it exists, is the most general solution: if \( \sigma = \text{unify}(C) \) and \( \sigma' \) is a solution to \( C \), then there is some \( \sigma'' \) such that \( \sigma' = (\sigma'' \circ \sigma) \).