Lecture 23
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

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

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

\[
\begin{align*}
\Delta, \Gamma & \vdash n : \text{int} \\
\Delta, \Gamma & \vdash x : \tau \\
\Delta, \Gamma, x : \tau & \vdash e : \tau' \quad \Delta \vdash \tau \ \text{ok} \\
\Delta, \Gamma & \vdash \lambda x : \tau. e : \tau \to \tau' \\
\Delta \cup \{X\}, \Gamma & \vdash e : \tau \\
\Delta, \Gamma & \vdash \forall X. e : \forall X. \tau \\
\Gamma(x) = \tau & \\
\Delta, \Gamma & \vdash \text{ok} \\
\Delta, \Gamma & \vdash e_1 : \tau \to \tau' \\
\Delta, \Gamma & \vdash e_2 : \tau \\
\Delta, \Gamma & \vdash e_1 \ e_2 : \tau' \\
\Delta, \Gamma & \vdash e_1 \ e_2 : \tau' \\
\Delta \vdash \tau \ \text{ok} \\
\Delta, \Gamma & \vdash e \ [\tau] : \tau' \{\tau/X\}
\end{align*}
\]
Polymorphism let us write a doubling function that works for any type of function:

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

The type of this expression is:

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

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

For example, we can write:

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

and OCaml will figure out that the type is:

$\text{('a } \rightarrow \text{'a)} \rightarrow \text{'a } \rightarrow \text{'a}$

which is equivalent to the same System F type:

$\forall A. (A \rightarrow A) \rightarrow A \rightarrow A$
Type Inference

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

We can also write

```
double (fun x \to x+1) 7
```

and OCaml will infer that the polymorphic function `double` is instantiated at the type `int`.
ML Polymorphism

However, polymorphism in OCaml and other MLs, has some restrictions to ensure that type inference remains *decidable*.
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.
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: $\forall \alpha. \alpha \to \alpha$
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: $(\forall \alpha. \alpha \to \alpha) \to \text{int}$
ML Polymorphism

However, polymorphism in OCaml and other MLs, has some restrictions to ensure that type inference remains \textit{decidable}.

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

\textbf{Examples}

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

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
Example

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!

```ocaml
# 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}
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
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 \( b + 1 \)
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}
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 \( b + 1 \)
- 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 \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}
- the type of \( c \) must be the same as \( b \)

Putting all these pieces together:
\[ \lambda a : \texttt{int} \rightarrow \texttt{bool}. \lambda b : \texttt{int}. \lambda c : \texttt{int}. \text{if } a \ (b + 1) \text{ then } b \text{ else } c \]
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.
STLC for Type Inference

Let’s define the type inference judgment for this STLC language:

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

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

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

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

\[ \Gamma \vdash x : \tau \mid \emptyset \] CT-VAR
Constraint-Based Typing Judgment

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

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

\[ \Gamma \vdash n : \text{int} | \emptyset \quad \text{CT-INT} \]
Constraint-Based Typing Judgment

\[
\begin{align*}
\Gamma(x) &= \tau \\
\Gamma \vdash x : \tau & \quad \text{CT-VAR} \\
\Gamma \vdash n : \text{int} & \quad \text{CT-INT} \\
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash e_1 + e_2 : \text{int} & \quad \Gamma \vdash e_1 : \tau_1, \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash e_1 + e_2 : \text{int} & \quad \Gamma \vdash e_1 : \tau_1 = \text{int}, \Gamma \vdash e_2 : \tau_2 = \text{int} \\
\Gamma \vdash e_1 + e_2 : \text{int} & \quad \Gamma \vdash e_1 : \tau_1 = \text{int}, \Gamma \vdash e_2 : \tau_2 = \text{int} \\
\end{align*}
\]
Constraint-Based Typing Judgment

$$\Gamma(x) = \tau \quad \text{CT-VAR}$$

$$\Gamma \vdash x : \tau \mid \emptyset$$

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

$$\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2$$

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

$$\Gamma, x : \tau_1 \vdash e : \tau_2 \mid C$$

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

\[
\begin{align*}
\Gamma(x) &= \tau & \text{CT-VAR} \\
\Gamma \vdash x : \tau & \mid \emptyset & \text{CT-VAR} \\
\Gamma \vdash n : \text{int} & \mid \emptyset & \text{CT-INT} \\
\Gamma \vdash e_1 : \tau_1 & \mid C_1 & \Gamma \vdash e_2 : \tau_2 & \mid C_2 \\
\Gamma \vdash e_1 + e_2 : \text{int} & \mid C_1 \cup C_2 \cup \{\tau_1 = \text{int}, \tau_2 = \text{int}\} & \text{CT-ADD} \\
\Gamma, x : \tau_1 \vdash e : \tau_2 & \mid C & \text{CT-ABS} \\
\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \to \tau_2 & \mid C & \text{CT-ABS} \\
\Gamma \vdash e_1 : \tau_1 & \mid C_1 & \Gamma \vdash e_2 : \tau_2 & \mid C_2 \\
X \text{ fresh} & \quad C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to X\} & \text{CT-APP} \\
\Gamma \vdash e_1 \, e_2 : X & \mid C' & \text{CT-APP}
\end{align*}
\]
Solving Constraints

A *type substitution* is a finite map from type variables to types.

**Example:** The substitution

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

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

We can define substitution of type variables formally:

\[
(X) \equiv \begin{cases} 
& (X) \text{ if } X \in \text{int} \setminus \{0\} \\
& \text{int} \setminus \{0\} \text{ if } X \text{ not in the domain of } \text{int}
\end{cases}
\]
Type Substitution

We can define substitution of type variables formally:

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

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

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

\[
\sigma(\tau \rightarrow \tau') \triangleq \sigma(\tau) \rightarrow \sigma(\tau')
\]
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 $C$. 
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.
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

Unification Algorithm

\[
\text{unify}(\emptyset) \equiv \emptyset
\]

\[
\text{unify}(f \neq g [C]) \equiv
\begin{cases} 
\text{if } f = g & \text{then unify}(C) \\
\text{elseif } f = X \text{ and } X \text{ not a free variable of } g & \text{then unify}(C, f \neq X \circ [X \neq g]) \\
\text{elseif } g = X \text{ and } X \text{ not a free variable of } f & \text{then unify}(C, f \neq X \circ [X \neq g]) \\
\text{elseif } f = o_1 \text{ and } g = o_1 & \text{then unify}(C, f_0 = o_0 ; 1 = o_1 g) \\
\text{else fail}
\end{cases}
\]
Unification Algorithm

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

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

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

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

\[ unify(C') \]
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

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

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

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

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

\[
\text{unify}(\{\tau = \tau'\} \cup C') \triangleq
\begin{align*}
\text{if } \tau = \tau' \text{ then} & \quad \text{unify}(C') \\
\text{elseif } \tau = X \text{ and } X \text{ not a free variable of } \tau' \text{ then} & \quad \text{unify}(C'\{\tau'/X\}) \circ [X \mapsto \tau'] \\
\text{elseif } \tau' = X \text{ and } X \text{ not a free variable of } \tau \text{ then} & \quad \text{unify}(C'\{\tau/X\}) \circ [X \mapsto \tau] \\
\text{else if } \tau = \tau_0 \rightarrow \tau_1 \text{ and } \tau' = \tau'_0 \rightarrow \tau'_1 \text{ then} & \quad \text{unify}(C' \cup \{\tau_0 = \tau'_0, \tau_1 = \tau'_1\}) \\
\text{else} & \quad \text{fail}
\end{align*}
\]
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 \( \sigma' = (\sigma'' \circ \sigma) \).