Lecture 23
Type Inference

26 October 2016
Announcements

- HW #6 due tonight at 11:59pm
  We made one problem easier! Please see Piazza.

- HW #7 out now

- My office hours: Thursday instead of Friday (2–3pm)
Review: Polymorphic $\lambda$-Calculus

Syntax

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

$$
\nu ::= n \mid \lambda x : \tau. e \mid \Lambda X. e
$$

Dynamic Semantics

$$
E ::= [\cdot] \mid E e \mid \nu E \mid E [\tau]
$$

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

\[ \forall X. \, X \rightarrow X \]

**Type Rules**

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

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

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

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

**Type Inference**

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

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

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

\[ \Delta, \Gamma \vdash e \left[ \tau \right] : \tau' \{	au / X\} \]

\[ \forall X. \, X \rightarrow \text{int} \]
Review: Polymorphic $\lambda$-Calculus

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:

$$\text{double [int]} (\lambda n : \text{int. } n + 1) 7$$
Type Inference

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:

$\langle \mathbb{a} \rightarrow \mathbb{a} \rangle \rightarrow \mathbb{a} \rightarrow \mathbb{a}$

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

\[
\text{double (fun } x \rightarrow x+1) 7
\]

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*. 
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 ∀s may only appear in the “outermost” position.

Examples

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

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

**Examples**

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

These restrictions mean that certain terms that are typeable in System F are not typeable in ML!

Ocaml version 4.01.0

# fun x -> x 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

\[ \alpha \rightarrow \beta \]
\[ [\alpha \rightarrow \beta] \]
\[ (\alpha \rightarrow \beta) \rightarrow \beta \]
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:
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 \(b + 1\)
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}
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 \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 \( b \)

Putting all these pieces together:
\[ \lambda a : \textbf{int} \rightarrow \textbf{bool}. \lambda b : \textbf{int}. \lambda c : \textbf{int}. \text{if } a (b + 1) \text{ then } b \text{ else } c \]
Constraint-Based Inference

Let’s automate type inference!
Constraint-Based 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.
Constraint-Based 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.

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.

$$X = Y \rightarrow \text{int} \quad Y = \text{bool}$$
STLC for Type Inference

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

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

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

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

\[
\frac{x: \tau \in \Gamma}{\Gamma \vdash x : \tau \mid \emptyset} \quad \text{CT-VAR}
\]
Constraint-Based Typing Judgment

\[
\begin{align*}
\frac{\chi: \tau \in \Gamma}{\Gamma \vdash \chi: \tau \mid \emptyset} & \quad \text{CT-VAR} \\
\frac{\Gamma \vdash n: \text{int} \mid \emptyset}{\Gamma \vdash \text{int} \mid \emptyset} & \quad \text{CT-INT}
\end{align*}
\]
Constraint-BasedTyping Judgment

\[ \frac{\chi : \tau \in \Gamma}{\Gamma \vdash \chi : \tau \mid \emptyset} \quad \text{CT-VAR} \]

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

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

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

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

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

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

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

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

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

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

\[
\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad X \text{ fresh}}{\Gamma \vdash e_1 \ e_2 : X \mid C'} \quad \text{(CT-APP)}
\]
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:
Type Substitution

We can define substitution of type variables formally:

$$\sigma(\tau) \rightarrow \tau$$

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

We can define substitution of type variables formally:

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

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

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

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

\[
\sigma(\tau \rightarrow \tau') = \sigma(\tau) \rightarrow \sigma(\tau')
\]
Type Substitution

We can define substitution of type variables formally:

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

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

\[
\sigma(\tau \to \tau') = \sigma(\tau) \to \sigma(\tau')
\]

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:

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

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

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

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 $\tau = \tau'$.

\[ \tau_1 = \text{int} \]
\[ \tau_2 = \text{int} \]
Unification

Our constraints are of the form \( \tau \equiv \tau' \).

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

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

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

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

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

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

\[ unify(C') \]
Unification Algorithm

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

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

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

\[ unify(\emptyset) = [] \] (the empty substitution)

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

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

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

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

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

\[
\text{unify}(\{\tau = \tau'\} \cup C') =
\]
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 \(\tau = \tau_o \rightarrow \tau_1\) and \(\tau' = \tau_o' \rightarrow \tau_1'\) then

\[
\text{unify}(C' \cup \{\tau_0 = \tau_o', \tau_1 = \tau_1'\})
\]
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 $\sigma = unify(C)$ and $\sigma'$ is a solution to $C$, then there is some $\sigma''$ such that $\sigma' = (\sigma'' \circ \sigma)$.

\[ \lambda x. x \]
\[ \text{int} \rightarrow \text{int} \]