Review: Polymorphic \( \lambda \)-Calculus

Syntax

\[
\begin{align*}
e & ::= n \mid x \mid \lambda x: \tau. e \mid e_1 e_2 \mid \Lambda \alpha. e \mid e[\tau] \\
v & ::= n \mid \lambda x: \tau. e \mid \Lambda \alpha. e
\end{align*}
\]

Dynamic Semantics

\[
\begin{align*}
E & ::= [\cdot] \mid E e \mid v E \mid E[\tau] \\
e & \rightarrow e' \\
E[e] & \rightarrow E[e'] \\
(\lambda x: \tau. e) v & \rightarrow e\{v/x\} \\
(\Lambda \alpha. e)[\tau] & \rightarrow e\{\tau/\alpha\}
\end{align*}
\]
Review: Polymorphic $\lambda$-Calculus

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

\[
\text{double} \triangleq \Lambda\alpha. \lambda f: \alpha \to \alpha. \lambda x: \alpha. f(fx).
\]

The type of this expression is:

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

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 $\forall \alpha. \tau$ or e $[\tau]$. 
Type Inference

In languages like OCaml, programmers don’t have to annotate their programs with $\forall \alpha. \tau$ 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:

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

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 $\forall \alpha. \tau$ 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`.
Type Inference, Formally

The *type inference* (or *type reconstruction*) problem asks whether, for a given untyped \( \lambda \)-calculus expression \( e' \) there exists a well-typed System F expression \( e \) such that \( \text{erase}(e) = e' \).
Type Inference, Formally

The *type inference* (or *type reconstruction*) problem asks whether, for a given untyped \( \lambda \)-calculus expression \( e' \) there exists a well-typed System F expression \( e \) such that \( \text{erase}(e) = e' \)

It was shown to be **undecidable** by Wells in 1994.
Polymorphism in OCaml (and other MLs) has some restrictions to ensure that type inference remains decidable.
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.
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

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

Polymorphism in OCaml (and other MLs) has some restrictions to ensure that type inference remains 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!
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;;
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 λ-calculus and OCaml, but it is possible for the simply-typed λ-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 \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 \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 \( b \)
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:
\[ \lambda a : \textbf{int} \rightarrow \textbf{bool}. \lambda b : \textbf{int}. \lambda c : \textbf{int}. \text{if } a (b + 1) \text{ then } b \text{ else } c \]
Let’s automate type inference!

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 typesubstitution $\sigma$ that, for each equation, makes both sides syntactically equal.
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 \textit{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 \textit{type substitution} \( \sigma \) that, for each equation, makes both sides syntactically equal.
Let’s define the type inference judgment for this STLC language:

\[
\begin{align*}
e & ::= x \mid \lambda x : \tau. e \mid e_1 e_2 \mid n \mid e_1 + e_2 \\
\tau & ::= \textbf{int} \mid X \mid \tau_1 \rightarrow \tau_2
\end{align*}
\]

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

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

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

\[ \Gamma \vdash x : \tau \mid \emptyset \]

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

\[ \text{CT-ADD} \]

\[ \Gamma \vdash e_1 + e_2 : \text{int} \mid \emptyset \]

\[ \text{CT-ABS} \]

\[ \Gamma \vdash e_1 : \tau_1 \mid \emptyset \]

\[ \Gamma \vdash e_2 : \tau_2 \mid \emptyset \]

\[ \lambda x : \tau_1. e_1 : \tau_1 \]

\[ X \text{ fresh} \]

\[ C' = C_1[ C_2[ f \tau_1 = \tau_2 ] ] \]

\[ \text{CT-APP} \]

\[ \Gamma \vdash e_1 e_2 : X \mid \emptyset \]
Constraint-Based Typing Judgment

\[ \Gamma(x) = \tau \]
\[ \Gamma \vdash x : \tau \mid \emptyset \]  \hspace{1cm} CT-VAR

\[ \Gamma \vdash n : \text{int} \mid \emptyset \]  \hspace{1cm} CT-INT

\[ \Gamma \vdash e_1 : \tau_1 \mid C_1 \hspace{1cm} \Gamma \vdash e_2 : \tau_2 \mid C_2 \]
\[ \Gamma \vdash e_1 + e_2 : \text{int} \mid C_1 \cup C_2 \cup \{ \tau_1 = \text{int}, \tau_2 = \text{int} \} \]  \hspace{1cm} CT-ADD
Constraint-Based Typing Judgment

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

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

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

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

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

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

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

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

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

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

\[
\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad X \text{ fresh} \quad C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \rightarrow X\} \quad \text{CT-APP}
\]

\[
\Gamma \vdash e_1 e_2 : X \mid C'
\]
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:

\[ σ(X) \buildrel ≜ \over = \begin{cases} τ & \text{if } X \neq τ \\ σ(X) & \text{if } X \text{ not in the domain of } σ \end{cases} \]

\[ σ(int) \buildrel ≜ \over = \text{int} \]

\[ σ(τ → τ') \buildrel ≜ \over = σ(τ) → σ(τ') \]

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

\[ \sigma(X) \triangleq \begin{cases} 
\tau & \text{if } X \mapsto \tau \in \sigma \\
X & \text{if } X \text{ not in the domain of } \sigma 
\end{cases} \]
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.”
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 $\tau = \tau'$. 
Unification

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

\[
\text{unify} (\tau; \tau') \equiv \begin{cases} 
\text{theemptysubstitution} & \text{if } \tau = \tau' \\
\text{unify} (\text{C'} \tau / \text{X} \cdot \text{g}) \left[ \text{X} \mapsto \tau' \right] & \text{if } \tau = \text{X} \text{ and X is a free variable of } \tau' \\
\text{unify} (\text{C'} \cdot \tau / \text{X} \cdot \text{g}) \left[ \text{X} \mapsto \tau' \right] & \text{if } \tau' = \text{X} \text{ and X is a free variable of } \tau \\
\text{unify} (\text{C'} \cdot \tau_0 = \tau_0' , \tau_1 = \tau_1') \cdot \text{g} & \text{if } \tau = \tau_0! \tau_1 \text{ and } \tau' = \tau_0'! \tau_1' \\
\text{fail} & \text{otherwise}
\end{cases}
\]
Unification Algorithm

\[ \text{unify}(\emptyset) \triangleq [] \] (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
\[
\text{unify}(C'\{\tau'/X\}) \circ [X \mapsto \tau']
\]
unify(∅) ≜ [] (the empty substitution)

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

\[ \text{unify}(\emptyset) \triangleq [] \] (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 \( \tau = \tau_o \to \tau_1 \) and \( \tau' = \tau'_o \to \tau'_1 \) 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 \( \sigma = \text{unify}(C) \) and \( \sigma' \) is a solution to \( C \), then there is some \( \sigma'' \) such that \( \sigma' = (\sigma'' \circ \sigma) \).