Announcements

• PS 6 due today

• PS 7 out today
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 $\epsilon [\tau]$

For example, we can write

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

and OCaml will figure out that the type is

$\langle 'a \to 'a \rangle \to 'a \to 'a$

which is equivalent to the System F type:

$\forall A. (A \to A) \to A \to A$
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

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

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.
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 as top-most constructors in a type.
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 as top-most constructors in a type.

**Examples**
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 as top-most constructors in a type.

**Examples**

- Prenex: $\forall \alpha. \alpha \rightarrow \alpha$
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 as top-most constructors in a type.

**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 \textit{prenex polymorphism}, stipulate that $\forall$s may only appear as top-most constructors in a type.

**Examples**

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

These restrictions have the following practical ramifications:
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 as top-most constructors in a type.

**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
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 as top-most constructors in a type.

**Examples**

- Prenex: ∀α. α → α
- Not prenex: (∀α. α → α) → 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 typable in System F are not typable in ML!
Example

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

```
OCCaml 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
```
In the simply-typed lambda calculus, we explicitly annotate the type of function arguments: $\lambda x : \tau. e$
In the simply-typed lambda calculus, we explicitly annotate the type of function arguments: $\lambda x : \tau. e$

These annotations are used in the typing rule for functions
Type Inference

In the simply-typed lambda calculus, we explicitly annotate the type of function arguments: \( \lambda x : \tau. e \)

These annotations are used in the typing rule for functions.

Suppose that we didn’t want to provide type annotations for function arguments... We would need to guess a \( \tau \) to put into the type context!
Type Inference

In the simply-typed lambda calculus, we explicitly annotate the type of function arguments: $\lambda x : \tau. \; e$

These annotations are used in the typing rule for functions

Suppose that we didn’t want to provide type annotations for function arguments... We would need to guess a $\tau$ to put into the type context!

Can we still type check our program without these type annotations? For the simply typed-lambda calculus (and many of the extensions we have considered so far), the answer is yes: we can *infer* (or *reconstruct*) the types of a program
Example

Consider the following program:

\[ \lambda a. \lambda b. \lambda c. \text{if } a (b + 1) \text{ then } b \text{ else } c \]
Consider the following program:

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

Informal inference:
Example

Consider the following 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

Consider the following 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

Consider the following 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

Consider the following program:

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

Informal inference:

- \(b\) must be \textit{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 \textit{bool}
Example

Consider the following program:

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

Informal inference:

- \( b \) must be \textit{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 \textit{bool}
- the type of \( c \) must be the same as \( b \)
Example

Consider the following program:

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

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:

\[ \text{\(\lambda a : \textbf{int} \rightarrow \textbf{bool}. \lambda b : \textbf{int}. \lambda c : \textbf{int}. \text{if } a (b + 1) \text{ then } b \text{ else } c\)} \]
To automate type inference, we introduce a judgment

$$\Gamma \vdash e : \tau \mid C$$
To automate type inference, we introduce a judgment

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

Given a typing context $\Gamma$ and an expression $e$, it generates a set of constraints—equations between type
To automate type inference, we introduce a judgment

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

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

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, when applied to each equation makes the types syntactically equal.
To automate type inference, we introduce a judgment
\[ \Gamma \vdash e : \tau \mid C \]

Given a typing context \(\Gamma\) and an expression \(e\), it generates a set of constraints—equations between type

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, when applied to each equation makes the types syntactically equal.

In what follows, we’ll work with the following language

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

\[
\tau ::= \text{int} \mid X \mid \tau_1 \to \tau_2
\]
Constraint-Based Typing Judgment

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

\[
\begin{align*}
\frac{x: \tau \in \Gamma}{\Gamma \vdash x: \tau \mid \emptyset} & \quad \text{CT-Var} \\
\frac{\Gamma \vdash n: \text{int} \mid \emptyset}{\quad \text{CT-Int}}
\end{align*}
\]
Constraint-Based Typing Judgment

\[
\Gamma \vdash x : \tau \quad \text{CT-Var}
\]

\[
\Gamma \vdash n : \text{int} \quad \text{CT-Int}
\]

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

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

\[
\frac{x: \tau \in \Gamma}{\Gamma \vdash x: \tau \mid \emptyset} \quad \text{CT-Var} \\
\frac{\Gamma \vdash n: \text{int} \mid \emptyset}{\quad \text{CT-Int}}
\]

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

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

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

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

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

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

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

Example: the substitution

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

maps type variable \( X \) to \text{int} and \( Y \) to \text{int} \rightarrow \text{int}.
Solving Constraints

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

Example: the substitution

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

maps type variable \( X \) to \textbf{int} and \( Y \) to \textbf{int} \( \rightarrow \textbf{int} \)

Note that the same variable may occur in both the domain and range of a substitution. In that case, the intention is that the substitutions are performed simultaneously.
Solving Constraints

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

**Example:** the substitution

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

maps type variable \( X \) to \( \texttt{int} \) and \( Y \) to \( \texttt{int} \rightarrow \texttt{int} \).

Note that the same variable may occur in both the domain and range of a substitution. In that case, the intention is that the substitutions are performed simultaneously.

**Example:** the substitution

\[ X \mapsto \texttt{int}, \ Y \mapsto (\texttt{int} \rightarrow X) \]

maps \( Y \) to \( \texttt{int} \rightarrow X \).
Formally, we define substitution of type variables as follows:
Type Substitution

Formally, we define substitution of type variables as follows:

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

Formally, we define substitution of type variables as follows:

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

\[ \sigma(\text{int}) = \text{int} \]
Formally, we define substitution of type variables as follows:

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

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

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

Formally, we define substitution of type variables as follows:

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

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

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

Note that we don’t need to worry about avoiding variable capture, since there are no binders in the language of types.
Type Substitution

Formally, we define substitution of type variables as follows:

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

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

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

Note that we don’t need to worry about avoiding variable capture, since there are no binders in the language of types.

Given two substitutions \(\sigma\) and \(\sigma'\), we write \(\sigma \circ \sigma'\) for their composition: \((\sigma \circ \sigma')(\tau) = \sigma(\sigma'(\tau))\).
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 $\sigma(\tau) = \sigma(\tau')$
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 $C$
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 $C$.

So to solve a set of constraints $C$, we need to find a substitution that unifies $C$. 
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 $C$

So to solve a set of constraints $C$, we need to find a substitution that unifies $C$

If $\Gamma \vdash e : \tau \mid C$ and a solution for $C$ is $\sigma$, then $e$ has type $\tau'$ under $\Gamma$, where $\sigma(\tau) = \tau'$. On the other hand, if there are no substitutions that satisfy $C$, then $e$ is not typeable
Unification

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

\[ \text{unify}(\{\tau = \tau'\} \cup C') = \begin{cases} 
\text{unify}(C') & \text{if } \tau = \tau' \\
\text{unify}(C'\{\tau'/X\}) \circ [X \mapsto \tau'] & \text{else if } \tau = X \text{ and } X \text{ not a free variable of } \tau' \\
\text{unify}(C'\{\tau/X\}) \circ [X \mapsto \tau] & \text{else if } \tau' = X \text{ and } X \text{ not a free variable of } \tau \\
\text{unify}(C' \cup \{\tau_0 = \tau'_0, \tau_1 = \tau'_1\}) & \text{else if } \tau = \tau_0 \rightarrow \tau_1 \text{ and } \tau' = \tau'_0 \rightarrow \tau'_1 \\
\text{fail} & \text{else}
\end{cases} \]
Unification Properties

The check that $X$ is not a free variable of the other type ensures that the algorithm doesn’t produce a cyclic substitution (e.g., $X \mapsto (X \rightarrow X)$), which doesn’t make sense with our finite types.
The check that $X$ is not a free variable of the other type ensures that the algorithm doesn’t produce a cyclic substitution (e.g., $X \mapsto (X \rightarrow X)$), which doesn’t make sense with our finite types.

The unification algorithm always terminates.
The check that $X$ is not a free variable of the other type ensures that the algorithm doesn’t produce a cyclic substitution (e.g., $X \mapsto (X \to X)$), which doesn’t make sense with our finite types.

The unification algorithm always terminates.

Moreover, 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 $\sigma' = (\sigma'' \circ \sigma)$.