CS 4110

Programming Languages & Logics

Lecture 24
Type Inference

29 October 2014

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 $e\ [\tau]$

For example, we can write

```
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 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

```
double (fun x → 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

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

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: $\forall \alpha.\, \alpha \to \alpha$

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 as top-most constructors in a type

Examples

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

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 ∀s may only appear as top-most constructors in a type

Examples

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

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 ∀s may only appear as top-most constructors in a type

Examples

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

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: $\forall\alpha.\ \alpha \to \alpha$
- Not prenex: $(\forall\alpha.\ \alpha \to \alpha) \to \textbf{int}$

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!

```
        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

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

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

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 τ 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 τ 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:

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

Example

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 **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 **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 **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 **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 **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 **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 **bool**
- the type of c must be the same as b

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 **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 **bool**
- the type of c must be the same as b

Putting all these pieces together:

$\lambda a{:}\,\mathbf{int} \to \mathbf{bool}.\ \lambda b{:}\,\mathbf{int}.\ \lambda c{:}\,\mathbf{int}.\ \text{if } a\ (b+1) \text{ then } b \text{ else } c$

Constriant-Based Inference

To automate type inference, we introduce a judgment

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

Constriant-Based Inference

To automate type inference, we introduce a judgment

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

Given a typing context Γ and an expression e, it generates a set of *constraints*—equations between type

Constriant-Based Inference

To automate type inference, we introduce a judgment

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

Given a typing context Γ 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 Γ

A solution to a set of constarints is a *type substitution* σ that, when applied to each equation makes the types syntactically equal.

Constriant-Based Inference

To automate type inference, we introduce a judgment

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

Given a typing context Γ 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 Γ

A solution to a set of constarints is a *type substitution* σ that, when applied to each equation makes the types syntactically equal.

In what follows, we'll work with the following langauge

$$e ::= x \mid \lambda x : \tau .\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2$$

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

Constraint-Based Typing Judgment

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

Constraint-Based Typing Judgment

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \mid \emptyset}\ \text{CT-Var} \qquad\qquad \frac{}{\Gamma \vdash n : \textbf{int} \mid \emptyset}\ \text{CT-Int}$$

Constraint-Based Typing Judgment

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \mid \emptyset} \text{ CT-Var} \qquad \frac{}{\Gamma \vdash n : \textbf{int} \mid \emptyset} \text{ CT-Int}$$

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

Constraint-Based Typing Judgment

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau \mid \emptyset}\ \text{CT-Var} \qquad \frac{}{\Gamma \vdash n:\textbf{int} \mid \emptyset}\ \text{CT-Int}$$

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

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2 \mid C}{\Gamma \vdash \lambda x:\tau_1.\, e:\tau_1 \to \tau_2 \mid C}\ \text{CT-Abs}$$

Constraint-Based Typing Judgment

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \mid \emptyset}\ \text{CT-Var} \qquad \frac{}{\Gamma \vdash n : \mathbf{int} \mid \emptyset}\ \text{CT-Int}$$

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \mid C}{\Gamma \vdash \lambda x : \tau_1 .\, e : \tau_1 \to \tau_2 \mid C}\ \text{CT-Abs}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \\ X \text{ fresh} \quad C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to X\}\end{array}}{\Gamma \vdash e_1\ e_2 : X \mid C'}\ \text{CT-App}$$

Solving Constraints

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

$$[X \mapsto \mathbf{int}, Y \mapsto \mathbf{int} \to \mathbf{int}]$$

maps type variable X to **int** and Y to $\mathbf{int} \to \mathbf{int}$

Solving Constraints

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

Example: the substitution

$$[X \mapsto \textbf{int}, Y \mapsto \textbf{int} \to \textbf{int}]$$

maps type variable X to **int** and Y to $\textbf{int} \to \textbf{int}$

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

$$[X \mapsto \textbf{int}, Y \mapsto \textbf{int} \to \textbf{int}]$$

maps type variable X to **int** and Y to $\textbf{int} \to \textbf{int}$

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.

Example: the substitution

$$[X \mapsto \textbf{int}, Y \mapsto (\textbf{int} \to X)]$$

maps Y to $\textbf{int} \to X$.

Type Substitution

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:

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

$$\sigma(\mathbf{int}) = \mathbf{int}$$

Type Substitution

Formally, we define substitution of type variables as follows:

$$
\begin{aligned}
\sigma(X) &= \begin{cases} \tau & \text{if } X \mapsto \tau \in \sigma \\ X & \text{if } X \text{ not in the domain of } \sigma \end{cases} \\
\sigma(\mathbf{int}) &= \mathbf{int} \\
\sigma(\tau \to \tau') &= \sigma(\tau) \to \sigma(\tau')
\end{aligned}
$$

Type Substitution

Formally, we define substitution of type variables as follows:

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

$$\sigma(\textbf{int}) = \textbf{int}$$

$$\sigma(\tau \to \tau') = \sigma(\tau) \to \sigma(\tau')$$

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:

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

$$\sigma(\mathbf{int}) = \mathbf{int}$$

$$\sigma(\tau \to \tau') = \sigma(\tau) \to \sigma(\tau')$$

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 σ and σ', we write $\sigma \circ \sigma'$ for their composition: $(\sigma \circ \sigma')(\tau) = \sigma(\sigma'(\tau))$.

Unification

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

Unification

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

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

Unification

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

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

We say that substitution σ *satisfies* (or *unifies*) set of constraints *C* if σ unifies every constraint in *C*

Unification

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

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

We say that substitution σ *satisfies* (or *unifies*) set of constraints C if σ 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 σ *unifies* constraint $\tau = \tau'$ if $\sigma(\tau) = \sigma(\tau')$

We say that substitution σ *satisfies* (or *unifies*) set of constraints *C* if σ 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 σ, then *e* has type τ' under Γ, where $\sigma(\tau) = \tau'$. On the other hand, if there are no substitutions that satisfy *C*, then *e* is not typeable

Unification

$$
\begin{aligned}
\mathit{unify}(\emptyset) &= [] \qquad \text{(the empty substitution)}\\
\mathit{unify}(\{\tau = \tau'\} \cup C') &= \text{if } \tau = \tau' \text{ then}\\
&\qquad \mathit{unify}(C')\\
&\quad \text{else if } \tau = X \text{ and } X \text{ not a free variable of } \tau' \text{ then}\\
&\qquad \mathit{unify}(C'\{\tau'/X\}) \circ [X \mapsto \tau']\\
&\quad \text{else if } \tau' = X \text{ and } X \text{ not a free variable of } \tau \text{ then}\\
&\qquad \mathit{unify}(C'\{\tau/X\}) \circ [X \mapsto \tau]\\
&\quad \text{else if } \tau = \tau_o \to \tau_1 \text{ and } \tau' = \tau'_o \to \tau'_1 \text{ then}\\
&\qquad \mathit{unify}(C' \cup \{\tau_0 = \tau'_0, \tau_1 = \tau'_1\})\\
&\quad \text{else}\\
&\qquad \mathit{fail}
\end{aligned}
$$

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., $X \mapsto (X \to X)$), which doesn't make sense with our finite types

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 unification algorithm always terminates.

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 unification algorithm always terminates.

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