CS 4110

Programming Languages & Logics

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 λ-Calculus

Syntax

$$
\begin{aligned}
e &::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \Lambda X.\, e \mid e\,[\tau] \\
v &::= n \mid \lambda x\!:\!\tau.\, e \mid \Lambda X.\, e
\end{aligned}
$$

Dynamic Semantics

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

$$
\frac{e \to e'}{E[e] \to E[e']} \qquad \frac{}{(\lambda x\!:\!\tau.\, e)\, v \to e\{v/x\}} \qquad \frac{}{(\Lambda X.\, e)\,[\tau] \to e\{\tau/X\}}
$$

Review: Polymorphic λ-Calculus

$\forall X.\ X \to X$ int → int unit → unit
(int → int) → (int → int)

$$\frac{}{\Delta, \Gamma \vdash n : \mathbf{int}} \qquad \frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x : \tau}$$

$$\frac{\Delta, \Gamma, x{:}\tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau'} \qquad \frac{\Delta, \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash e_1\, e_2 : \tau'}$$

$$\frac{\Delta \cup \{X\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda X.\, e : \forall X.\, \tau} \qquad \frac{\Delta, \Gamma \vdash e : \forall X.\, \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta, \Gamma \vdash e\, [\tau] : \tau'\{\tau/X\}}$$

$\forall X.\ X \to \text{int}$

Review: Polymorphic λ-Calculus

Polymorphism let us write a doubling function that works for *any* type of function:

$$\text{double} \triangleq \Lambda X.\, \lambda f{:}X \to X.\, \lambda x{:}X.\, f\,(f\,x).$$

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:

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

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:

```
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 same 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 $\forall$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 $\forall$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 ∀s may only appear in the "outermost" position.

Examples

- Prenex: $\forall \alpha.\, \alpha \to \alpha$
- Not prenex: $(\forall \alpha.\, \alpha \to \alpha) \to$ **int**
- Not prenex: $(\forall \alpha.\, \alpha \to \alpha) \to$ **int**

$\forall \alpha.\ (\alpha \to \alpha) \to \mathrm{int}$

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 in the "outermost" position.

Examples

id : $\forall \alpha.\ \alpha \to \alpha$

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

id double

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
```

$\alpha \to \beta \quad [\alpha \to \beta]$

$(\alpha \to \beta) \to \beta$

Type Inference

Type inference may be undecidable for the polymorphic λ-calculus and OCaml, but it is possible for the simply-tpyed λ-calculus!

Type Inference

Type inference may be undecidable for the polymorphic λ-calculus and OCaml, but it is possible for the simply-tpyed λ-calculus!

Type inference for the STLC means guessing a τ 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:

$\lambda a.\ \lambda b.\ \lambda c.$ if $a\ (b + 1)$ then b else c

int

: int

: int

a function

int

int

int → bool

: int → 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:

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 **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 **int**
- *a* must be some kind of function

Example

Here's an untyped program:

$\lambda a.\, \lambda b.\, \lambda c.$ if $a\ (b + 1)$ then b 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

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 **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

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 **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

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 **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:

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

Constraint-Based Inference

Let’s automate type inference!

Constriant-Based Inference

Let's automate type inference!

We introduce a new judgment:

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

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

Constriant-Based Inference

Let's automate type inference!

We introduce a new judgment:

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

Given a typing context Γ 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 Γ.

A solution to a set of constraints is a *type substitution* σ that, for each equation, makes both sides syntactically equal.

$X = Y \to \text{int}$ $\quad$ $Y = \text{bool}$

STLC for Type Inference

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

$$e ::= x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid n \mid e_1 + e_2$$
$$\tau ::= \mathbf{int} \mid X \mid \tau_1 \rightarrow \tau_2$$

You can use a type variable X wherever you want to have a type inferred.

$\lambda x.\ x$

$\lambda x : \tau .\ x$

$\lambda x : T .\ x$

Constraint-Based Typing Judgment

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

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

Constraint-Based Typing Judgment

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

Constraint-Based Typing Judgment

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

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

Constraint-Based Typing Judgment

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

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

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

Constraint-Based Typing Judgment

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

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

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

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

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 $\mathbf{int} \to \mathbf{int}$.

Type Substitution

We can define substitution of type variables formally:

Type Substitution

We can define substitution of type variables formally:

$\sigma(\tau) \to \tau$

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

Type Substitution

We can define substitution of type variables formally:

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

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

Type Substitution

We can define substitution of type variables formally:

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

Type Substitution

We can define substitution of type variables formally:

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

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

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

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:

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

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

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

We don't need to worry about avoiding variable capture: all type variables are "free."

Given two substitutions σ_1 and σ_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 $\tau = \tau'$.

$\tau_1 = int$

$\tau_2 = int$

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

If:

- $\Gamma \vdash e : \tau \mid C$, and
- σ satisfies C,

then e has type τ' under Γ,
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
- σ satisfies C,

then e has type τ' under Γ,
where $\sigma(\tau) = \tau'$.

If there are no substitutions that satisfy C, then e is not typeable.

So let's find a substitution σ that unifies a set of constraints C!

Unification Algorithm

Unification Algorithm

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

Unification Algorithm

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

$\mathit{unify}(\{\tau = \tau'\} \cup C') =$
if $\tau = \tau'$ then
$\quad \mathit{unify}(C')$

Unification Algorithm

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

$\mathit{unify}(\{\tau = \tau'\} \cup C') =$
if $\tau = \tau'$ then
    $\mathit{unify}(C')$
else if $\tau = X$ and X not a free variable of τ' then
    $\mathit{unify}(C'\{\tau'/X\}) \circ [X \mapsto \tau']$

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 τ' then
  $unify(C'\{\tau'/X\}) \circ [X \mapsto \tau']$
else if $\tau' = X$ and X not a free variable of τ then
  $unify(C'\{\tau/X\}) \circ [X \mapsto \tau]$

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 τ' then
  $unify(C'\{\tau'/X\}) \circ [X \mapsto \tau']$
else if $\tau' = X$ and X not a free variable of τ then
  $unify(C'\{\tau/X\}) \circ [X \mapsto \tau]$
else if $\tau = \tau_o \to \tau_1$ and $\tau' = \tau_o' \to \tau_1'$ then
  $unify(C' \cup \{\tau_0 = \tau_0', \tau_1 = \tau_1'\})$

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 τ' then
    $unify(C'\{\tau'/X\}) \circ [X \mapsto \tau']$
else if $\tau' = X$ and X not a free variable of τ then
    $unify(C'\{\tau/X\}) \circ [X \mapsto \tau]$
else if $\tau = \tau_o \to \tau_1$ and $\tau' = \tau'_o \to \tau'_1$ then
    $unify(C' \cup \{\tau_0 = \tau'_0, \tau_1 = \tau'_1\})$
else
    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 $\sigma = \mathit{unify}(C)$ and σ' is a solution to C, then there is some σ'' such that $\sigma' = (\sigma'' \circ \sigma)$.

$\lambda x.\ x$

int → int