Type Inference III: In Practice

CAS CS 320: Principles of Programming Languages

November 26, 2024 (Lecture 23)

Practice Problem

$$\cdot \vdash \lambda f.\lambda x.f(x+1) : \tau \dashv \mathcal{C}$$

Determine the type τ and constraints $\mathcal{C}$ such that the above judgment is derivable in HM^{-}.

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv C_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \dashv C_2}{\Gamma \vdash \textsf{let } x = e_1 \textsf{ in } e_2 : \tau_2 \dashv C_1, C_2} \text{ (let)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \alpha \text{ is fresh}}{\Gamma \vdash e_2 : \alpha \dashv \tau_1 = \tau_2 \to \alpha, \mathcal{C}_1, \mathcal{C}_2} \text{ (app)} \qquad \frac{\alpha \text{ is fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \lambda x.e : \alpha \to \tau \dashv \mathcal{C}} \text{ (fun)}$$

Answer

$$\cdot \vdash \lambda f.\lambda x.f(x+1) : \tau \dashv \mathcal{C}$$

Today

- Look at an inference rule for recursive let-expressions
- Finish up our discussion on unification
- Demo an implementation of the unification algorithm and its use in type-inference

Learning Objectives

- Determine how a collection of constraints is unified
- Determine the principle type of an expression in HM $^-$ in a given context
- Determine if an expression in HM$^-$ is ill-typed (i.e., determine when unification fails)

Outline

Recap: Hindley-Milner (Light)

Unification

Recap: Syntax (Mathematical)

$$
\begin{array}{rcl}
e & ::= & \lambda x.e \mid ee \\
& \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \\
& \mid & \mathsf{let\ rec}\ f\ x = e\ \mathsf{in}\ e \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& \mid & e + e \mid e = e \\
& \mid & num \mid x \\
\sigma & ::= & \mathsf{int} \mid \mathsf{bool} \mid \sigma \to \sigma \mid \alpha \\
\tau & ::= & \sigma \mid \forall \alpha.\tau
\end{array}
$$

Recap: Type System (Basics)

monotype

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \mathsf{int} \dashv \varnothing}\ (\mathsf{int}) \qquad \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma \dashv \varnothing}\ (\mathsf{var})$$

constraints

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 + e_2 : \mathsf{int} \dashv \tau_1 = \mathsf{int}, \tau_2 = \mathsf{int}, \mathcal{C}_1, \mathcal{C}_2}\ (\mathsf{add})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \Gamma \vdash e_3 : \tau_3 \dashv \mathcal{C}_3}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau_3 \dashv \tau_1 = \mathsf{bool}, \tau_2 = \tau_3, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3}\ (\mathsf{if})$$

(*Exercise.* Write the rule for integer equality)

Recap: Type System (Functions and Variables)

$$\frac{\alpha \text{ is fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \lambda x.e : \alpha \to \tau \dashv \mathcal{C}} \ (\mathsf{fun})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \alpha \text{ is fresh}}{\Gamma \vdash e_1\, e_2 : \alpha \dashv \tau_1 = \tau_2 \to \alpha, \mathcal{C}_1, \mathcal{C}_2} \ (\mathsf{app})$$

$$\frac{(x : \forall \alpha_1 . \forall \alpha_2 \ldots \forall \alpha_k . \tau) \in \Gamma \qquad \beta_1, \ldots, \beta_k \text{ are fresh}}{\Gamma \vdash x : [\beta_1 / \alpha_1] \ldots [\beta_k / \alpha_k] \tau} \ (\mathsf{var})$$

allows for polymorphism

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \dashv \mathcal{C}_1, \mathcal{C}_2} \ (\mathsf{let})$$

not let-polymorphic

Type System (Recursive Let-Expressions)

Our last (interesting) rule:

$f \approx \lambda x . e_1$

$$\frac{\alpha, \beta \text{ are free} \qquad \Gamma, f : \alpha \to \beta, x : \alpha \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma, f : \alpha \to \beta \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \textsf{let rec } f \ x = e_1 \textsf{ in } e_2 : \tau_2 \dashv \boxed{\beta = \tau_1}, \mathcal{C}_1, \mathcal{C}_2} \ (\cancel{\text{let}}) \ (\text{letRec})$$

When we type a recursive function, we don't know *anything* about it, except that it is a function.

We do know after the fact that the output type β has to equal τ_1, the type of e_1 (the body of f).

Example

```
let rec f x = f (f (x + 1)) in f
```

$\cdot \vdash \text{let rec } f\ x = f(f\ (x+1))\ \text{in } f : \alpha \to \beta,\ C$

- $\{ f:\alpha\to\beta,\ x:\alpha \} \vdash f\ (f\ (x+1)) : \eta \dashv \alpha\to\beta = \gamma \to \eta, \ldots$ (app)
  - $\{ f:\alpha\to\beta,\ x:\alpha \} \vdash f : \alpha \to \beta \dashv \emptyset$ (var)
  - $\{ f:\alpha\to\beta,\ x:\alpha \} \vdash f\ (x+1) : \gamma \dashv \alpha\to\beta = int \to \gamma,\ \alpha = int,\ int = int$ (app)
    - $\{ \ldots \} \vdash f : \alpha \to \beta \dashv \emptyset$ (var)
    - $\{ \ldots \} \vdash x + 1 : int \dashv \alpha = int,\ int = int$ (add)
      - $\{ \ldots \} \vdash x : \alpha \dashv \emptyset$ (var)
      - $\{ \ldots \} \vdash 1 : int \dashv \emptyset$ (int)
- $\{ f:\alpha\to\beta \} \vdash f : \alpha \to \beta \dashv \emptyset$

$C = \beta = \eta,\ \boxed{\alpha} \to \beta = \boxed{\gamma} \to \eta,\ \alpha \to \boxed{\beta} = int \to \boxed{\gamma},\ \boxed{\alpha = int},\ int = int$

Outline

Recap: Hindley-Milner (Light)

Unification

Recap: Unification Problem

Informal. Given an ADT, we consider a **term** to be an element of the ADT possibly with variables (this can be made formal using ideas from logic and algebra)

e.g. α $\qquad$ $\alpha \to \beta$ $\qquad$ $\mathrm{int} \to \mathrm{int}$

A **unification problem** is a collection of equations of the form

$$
\begin{gathered}
s_1 \boxed{\doteq} t_1 \\
s_2 \doteq t_2 \\
\vdots \\
s_k \doteq t_k
\end{gathered}
\qquad \alpha \doteq \beta
$$

where $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$ are terms.

Recap:Type Unification

```
type ty =
  | TInt
  | TBool
  | TFun of ty * ty
  | TVar of string
```

Type unification is the particular unification problem over the `ty` ADT (with type variable acting as variables in the sense from the previous slide)

$$\mathrm{TFun}(\alpha, \beta) \doteq \mathrm{TFun}(\gamma, \mathrm{TInt})$$

$$\gamma \doteq \mathrm{TInt}$$

Recap: Unifiers

Given a unification problem $\mathcal{U}$, a **solution** or **unifier** is a sequence of substitutions to some of the variables which appear in $\mathcal{U}$, typically written

$$\mathcal{S} = \boxed{\{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n\}}$$

We then write $\mathcal{S}t$ for the term $[t_n/x_n] \ldots [t_1/x_1]t$

A solution has the property that it *satisfies* every equation

$$\begin{aligned} \mathcal{S}t_1 &= \mathcal{S}s_1 \\ \mathcal{S}s_2 &= \mathcal{S}t_2 \\ &\vdots \\ \mathcal{S}s_k &= \mathcal{S}t_k \end{aligned}$$

actual equality

syntactic

Most General Unifiers

The **most general unifier** of a unification problem is the solution $\mathcal{S}$ such that, for any other solution $\mathcal{S}'$, there is another solution $\mathcal{S}''$ such that

$$\mathcal{S}' = \mathcal{S}\mathcal{S}''$$

$S\alpha = [int/\beta]\alpha$

In other words, every other unifier is $\mathcal{S}$ with more substitutions

Our algorithm is guaranteed to return the most general unifier

$\alpha \doteq \alpha$

$\beta \doteq int$

$S\alpha = \boxed{\alpha = \alpha} = S\alpha$

$S\beta = \boxed{int = int} = S\ int$

$S = \{\beta \mapsto int\}$

is MGU

$S' = \{\beta \mapsto int, \alpha \mapsto int\}$

extra sub.

is not

Principle Types

$$\Gamma \vdash e : \tau \dashv \mathcal{C}$$

The constraints $\mathcal{C}$ define a unification problem

Given a unifier $\mathcal{S}$ for $\mathcal{C}$ can get the "actual" type of e, its τ *after the substition* $\mathcal{S}$, i.e., $\mathcal{S}\tau\}$

If $\mathcal{S}$ is the most general unifier then $\mathcal{S}\tau$ (after "polymorphization") is called the **principle type** of e. The principle type has the property that every other type is an *instance* of it

An Algorithm

The idea. We process equations one at a time, updating *the collection of equations themselves.* We **FAIL** if we every reach an unsatisfiable equation. There are three success cases:

Start with an empty solution $\mathcal{S}$. Given a type unification problem $\mathcal{U}$, if $\mathcal{U}$ has the equation:

- $t_1 \doteq t_2$ where $t_1 = t_2$ (they are syntactically equal, e.g. $\mathsf{int} \doteq \mathsf{int}$ or $\alpha \doteq \alpha$, then remove it from $\mathcal{U}$
- $s_1 \to s_2 \doteq t_1 \to t_2$, then remove it and add $s_1 \doteq t_1$ and $s_2 \doteq t_2$ to $\mathcal{U}$
- $\alpha \doteq t$ or $t \doteq \alpha$ where α does not appear free in t e.g. $\alpha \doteq \alpha \to \mathsf{int}$
  - remove it
  - add $\alpha \mapsto t$ to $\mathcal{S}$
  - perform the substitution $\alpha \mapsto t$ in *every* equation remaining in $\mathcal{U}$

  (this removes α)
- otherwise, **FAIL**

  e.g. $\mathsf{int} \doteq \mathsf{bool}$

Repeat until $\mathcal{U}$ is empty

Example (Familiar)

$\cancel{\mathrm{int} \doteq \beta}$ (term $\doteq$ var)

$\cancel{\mathrm{int} \to \mathrm{int} \doteq \mathrm{int} \to \delta}$ (fun $\doteq$ fun)

$\cancel{\mathrm{int} \doteq \mathrm{int}}$ (eq)

$\cancel{\mathrm{int} \doteq \delta}$ (term $\doteq$ var)

$\cancel{\alpha \doteq \delta \to \eta}$ (var $\doteq$ term)

$\cancel{\gamma \doteq \mathrm{int} \to \delta}$ (var $\doteq$ term)

$\cancel{\mathrm{int} \to (\mathrm{int} \to \mathrm{int}) \doteq \beta \to \cancel{\gamma}\,(\mathrm{int} \to \delta)}$ (fun $\doteq$ fun)

$$\boxed{S = \{\alpha \mapsto \delta \to \eta,\ \gamma \mapsto \mathrm{int} \to \delta,\ \beta \mapsto \mathrm{int},\ \delta \mapsto \mathrm{int}\}}$$

$$S(\alpha \to \beta \to \eta) = (\mathrm{int} \to \eta) \to \mathrm{int} \to \eta$$

Example (Familiar)

$$
\begin{aligned}
\alpha &\doteq \delta \to \eta \\
\gamma &\doteq \mathsf{int} \to \delta \\
\mathsf{int} \to \mathsf{int} \to \mathsf{int} &\doteq \beta \to \gamma
\end{aligned}
$$

$$\text{out} \doteq \alpha \to \beta \to \eta$$

Example (Familiar)

$\alpha \to \beta$

$\downarrow \beta \mapsto \eta$

$\alpha \to \eta$

$\downarrow \eta \mapsto \gamma$

$\alpha \to \gamma$

$\downarrow \alpha \mapsto \gamma$

$\gamma \to \gamma$

$\downarrow \gamma \mapsto \mathrm{int}$

$\mathrm{int} \to \mathrm{int}$

$\cancel{\beta \doteq \eta}$ (var $\doteq$ term)

$\cancel{\alpha \to \beta \doteq \alpha \to \gamma}$ (f $\doteq$ f)

$\cancel{\alpha \to \beta \doteq \gamma \to \eta}$ (f $\doteq$ f)

$\cancel{\alpha \to \beta \doteq \mathsf{int} \to \eta}$? (f $\doteq$ f)

$\cancel{\alpha \doteq \alpha}$ (eq)

$\cancel{\eta \doteq \gamma}$ (var $\doteq$ t)

$\cancel{\alpha \doteq \gamma}$ (v $\doteq$ t)

$\cancel{\eta \doteq \eta}$ ($\gamma \doteq \gamma$)

$\cancel{\alpha \doteq \mathrm{int}}$ ($\gamma \doteq \mathrm{int}$)

$\cancel{\eta \doteq \eta}$ ($\mathrm{int} \doteq \mathrm{int}$)

$S = \{\beta \mapsto \eta,\ \eta \mapsto \gamma,\ \alpha \mapsto \gamma,\ \gamma \mapsto \mathrm{int}\}$

Putting Everything Together

- Constrainted-based inference:

$$\cdot \vdash \lambda f.\lambda x.f(x+1) : \alpha \rightarrow \beta \rightarrow \eta \dashv \alpha \doteq \delta \rightarrow \eta, \gamma \doteq \mathsf{int} \rightarrow \delta, \\ \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int} \doteq \beta \rightarrow \gamma$$

- Unification:

$$\mathcal{S} = \{\alpha \mapsto \delta \rightarrow \eta, \gamma \mapsto \mathsf{int} \rightarrow \delta, \beta \mapsto \mathsf{int}\}$$
$$\mathcal{S}(\alpha \rightarrow \beta \rightarrow \eta) = (\mathsf{int} \rightarrow \eta) \rightarrow \mathsf{int} \rightarrow \eta$$

- Generalization:

$$\cdot \vdash \lambda f.\lambda x.f(x+1) : \forall \eta.(\mathsf{int} \rightarrow \eta) \rightarrow \mathsf{int} \rightarrow \eta$$

Example (Unification Failure)

$$\cdot \vdash \lambda x.xx : \tau \dashv \mathcal{C}$$