

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^- .

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv C_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv C_2 \quad \alpha \text{ is fresh}}{\Gamma \vdash e_2 : \alpha \dashv \tau_1 = \tau_2 \rightarrow \alpha, C_1, C_2} \text{ (app)} \qquad \frac{\alpha \text{ is fresh} \quad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \lambda x. e : \alpha \rightarrow \tau \dashv \mathcal{C}} \text{ (fun)}$$

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

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)

Recap: Hindley-Milner (Light)

Unification

Recap: Syntax (Mathematical)

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

Recap: Type System (Basics)

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

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

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

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

Recap: Type System (Functions and Variables)

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

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

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

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

Type System (Recursive Let-Expressions)

Our last (interesting) rule:

$$\frac{\alpha, \beta \text{ are free} \quad \Gamma, f : \alpha \rightarrow \beta, x : \alpha \vdash e_1 : \tau_1 \dashv C_1 \quad \Gamma, f : \alpha \rightarrow \beta \vdash e_2 : \tau_2 \dashv C_2}{\Gamma \vdash \text{let rec } f \ x = e_1 \text{ in } e_2 : \tau_2 \dashv \beta = \tau_1, C_1, C_2} \text{ (let)}$$

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
```

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](#))

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

$$\begin{array}{c} s_1 \doteq t_1 \\ s_2 \doteq t_2 \\ \vdots \\ s_k \doteq t_k \end{array}$$

where s_1, \dots, s_k and t_1, \dots, 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)

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

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

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

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

$$\mathcal{S}t_1 = \mathcal{S}s_1$$

$$\mathcal{S}s_2 = \mathcal{S}t_2$$

$$\vdots$$

$$\mathcal{S}s_k = \mathcal{S}t_k$$

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

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

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

Our algorithm is guaranteed to return the most general unifier

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

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 substitution* \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 ever 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. $\text{int} \doteq \text{int}$ or $\alpha \doteq \alpha$, then remove it from \mathcal{U})
- ▶ $s_1 \rightarrow s_2 \doteq t_1 \rightarrow t_2$, 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 does not appear free in t**
 - ▶ remove it
 - ▶ add $\alpha \mapsto t$ to \mathcal{S}
 - ▶ perform the substitution $\alpha \mapsto t$ in *every* equation remaining in \mathcal{U}
- ▶ otherwise, **FAIL**

Repeat until \mathcal{U} is empty

Example (Familiar)

$$\alpha \doteq \delta \rightarrow \eta$$

$$\gamma \doteq \text{int} \rightarrow \delta$$

$$\text{int} \rightarrow \text{int} \rightarrow \text{int} \doteq \beta \rightarrow \gamma$$

Example (Familiar)

$$\beta \doteq \eta$$

$$\alpha \rightarrow \beta \doteq \alpha \rightarrow \gamma$$

$$\alpha \rightarrow \beta \doteq \gamma \rightarrow \eta$$

$$\alpha \rightarrow \beta \doteq \mathbf{int} \rightarrow \eta$$

Putting Everything Together

- ▶ Constrained-based inference:

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

- ▶ Unification:

$$\begin{aligned} \mathcal{S} &= \{\alpha \mapsto \delta \rightarrow \eta, \gamma \mapsto \text{int} \rightarrow \delta, \beta \mapsto \text{int}\} \\ \mathcal{S}(\alpha \rightarrow \beta \rightarrow \eta) &= (\text{int} \rightarrow \eta) \rightarrow \text{int} \rightarrow \eta \end{aligned}$$

- ▶ Generalization:

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

Example (Unification Failure)

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