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^-.

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

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

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: Syntax (Mathematical)

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

Recap: Type System (Basics)

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

constraints

no assumption about types

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

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

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

Recap: Type System (Functions and Variables)

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

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

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

use polymorphism

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

no let-polymorphism

Type System (Recursive Let-Expressions)

Our last (interesting) rule:

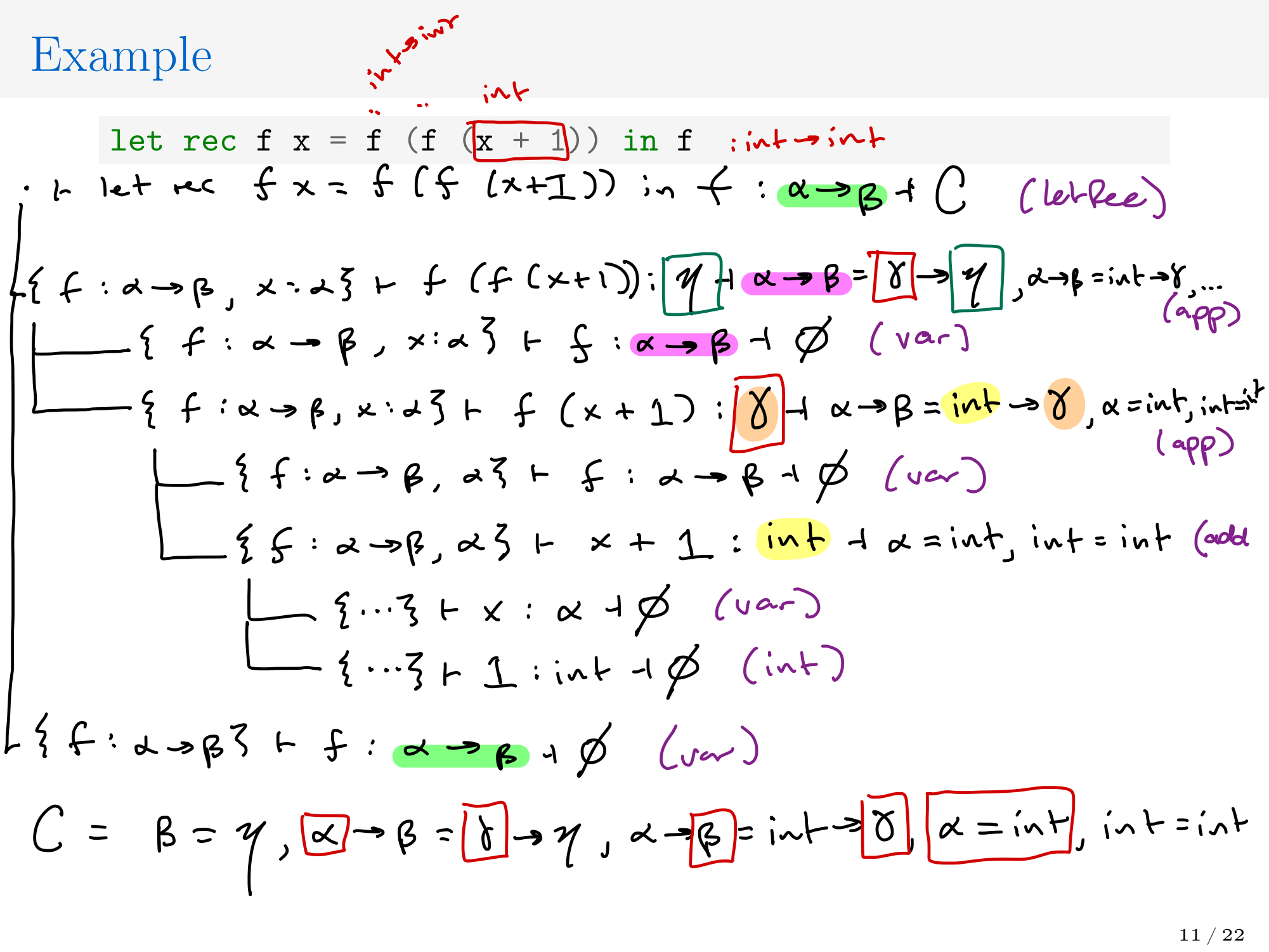
$$\frac{\alpha, \beta \text{ are free} \qquad \Gamma, f : \alpha \to \beta, x : \alpha \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma, f : \alpha \to \beta \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \mathsf{let\ rec}\ f\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \dashv \beta = \tau_1, \mathcal{C}_1,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

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  : int → int
```



Outline

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)

$\alpha = \beta \qquad \beta = \text{int} \to \text{int}$

term

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

can be unified

$$\begin{aligned} s_1 &\doteq t_1 \\ s_2 &\doteq t_2 \\ &\vdots \\ s_k &\doteq t_k \end{aligned}$$

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)

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

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

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

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

syntactic equality

$\alpha \doteq \beta$

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

$S\alpha = [\beta/\text{int}][\alpha/\text{int}]\alpha$

$= \text{int}$

$S\beta = \text{int}$ $\quad \text{int} = \text{int}$ ✓

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

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

+ some additional subs

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

Our algorithm is guaranteed to return the most general unifier

$\alpha \doteq \beta$

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

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

$S\alpha = \beta$

$S\beta = \beta$

$\beta = \beta$ ✓

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. $\alpha \doteq \mathsf{int} \to \alpha$)
  - remove it
  - add $\alpha \mapsto t$ to $\mathcal{S}$
  - perform the substitution $\alpha \mapsto t$ in *every* equation remaining in $\mathcal{U}$ (eliminate α in $\mathcal{U}$)
- otherwise, **FAIL** (e.g. $\mathsf{bool} \doteq \mathsf{int}$)

Repeat until $\mathcal{U}$ is empty

Example (Familiar)

$$\alpha \doteq \delta \to \eta$$
$$\gamma \doteq \mathsf{int} \to \delta$$
$$\mathsf{int} \to \mathsf{int} \to \mathsf{int} \doteq \beta \to \gamma$$

out $\doteq \alpha \to \beta \to \gamma$

Example (Familiar)

$S(\alpha \to \beta \to \eta)$

$\alpha \to \beta \to \eta$

$\downarrow \alpha \mapsto \delta \to \eta$

$(\delta \to \eta) \to \beta \to \eta$

$\downarrow \gamma \mapsto \text{int} \to \delta$

$(\delta \to \eta) \to \beta \to \eta$

$\downarrow \beta \mapsto \text{int}$

$(\delta \to \eta) \to \text{int} \to \eta$

$\downarrow \delta \mapsto \text{int}$

$(\text{int} \to \eta) \to (\text{int} \to \eta)$

$\alpha \to \beta \to \eta$

~~$\alpha \doteq \delta \to \eta$~~ $(v \doteq t)$

~~$\gamma \doteq \text{int} \to \delta$~~ $(v \doteq t)$

~~$\text{int} \to (\text{int} \to \text{int}) \doteq \beta \to \gamma (\text{int} \to \delta)$~~ $(f \doteq f)$

~~$\text{int} \doteq \beta$~~ $(t \doteq v)$

~~$\text{int} \to \text{int} \doteq \text{int} \to \delta$~~ $(f \doteq f)$

~~$\text{int} \doteq \text{int}$~~ (eq)

~~$\text{int} \doteq \delta$~~ $(t \doteq v)$

```
fun f -> fun x -> f (x+1)
```

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

Example (Familiar)

$\alpha \to \beta$

$\downarrow \beta \mapsto \eta$

$\alpha \to \eta$

$\downarrow \eta \mapsto \gamma$

$\alpha \to \gamma$

$\downarrow \alpha \mapsto \gamma$

$\gamma \to \gamma$

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

$\text{int} \to \text{int}$

~~$\beta \doteq \eta$~~ $(\mathrm{v} \doteq \mathrm{t})$

~~$\alpha \to \beta \doteq \alpha \to \eta$~~ $(\mathrm{f} \doteq \mathrm{f})$

~~$\alpha \to \beta \doteq \eta \to \eta$~~ $(\mathrm{f} \doteq \mathrm{f})$

~~$\alpha \to \beta \doteq \text{int} \to \eta$~~ $(\mathrm{f} \doteq \mathrm{f})$

~~$\alpha \doteq \alpha$~~ (eq)

~~$\eta \doteq \gamma$~~ $(\mathrm{v} \doteq \mathrm{t})$

~~$\alpha \doteq \gamma$~~ $(\mathrm{v} \doteq \mathrm{t})$

~~$\gamma \to \eta \doteq \eta \to \gamma$~~

~~$\gamma \to \alpha \doteq \text{int}$~~

~~$\text{int} \to \eta \doteq \eta \to \text{int}$~~

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

Putting Everything Together

- Constrainted-based inference:

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

- Unification:

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

- Generalization:

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

Example (Unification Failure)

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