

Specialization

Concepts of Programming Languages

Outline

- » Discuss **specialization** and how it relates to principle types
- » Demo an implementation of **constraint-based type inference**
- » Put the finishing touches on our discussion of type inference

Recap

Recall: Principle Types

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

Recall: Principle Types

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

The constraints \mathcal{C} defined a *unification problem*. Given a most general unifier \mathcal{S} we can get the "actual" type of e :

Recall: Principle Types

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

The constraints \mathcal{C} defined a *unification problem*. Given a most general unifier \mathcal{S} we can get the "actual" type of e :

$$\text{principle}(\tau, \mathcal{C}) = \forall \alpha_1 \dots \forall \alpha_k. \mathcal{S}\tau \text{ where } \text{FV}(\mathcal{S}\tau) = \{\alpha_1, \dots, \alpha_k\}$$

Recall: Principle Types

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

The constraints \mathcal{C} defined a *unification problem*. Given a most general unifier \mathcal{S} we can get the "actual" type of e :

$$\text{principle}(\tau, \mathcal{C}) = \forall \alpha_1 \dots \forall \alpha_k. \mathcal{S}\tau \text{ where } \text{FV}(\mathcal{S}\tau) = \{\alpha_1, \dots, \alpha_k\}$$

i.e, the **principle type** of e (note: it may not exist). Every type we *could* give e is a *specialization* of $\forall \alpha_1, \dots, \alpha_k. \mathcal{S}\tau$

Recall: Putting everything together

Recall: Putting everything together

input: program P (sequence of top-level let-expressions)

Recall: Putting everything together

input: program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \emptyset$

Recall: Putting everything together

input: program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \emptyset$

FOR EACH top-level let-expression $\text{let } x = e \text{ in } P$:

Recall: Putting everything together

input: program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \emptyset$

FOR EACH top-level let-expression $\text{let } x = e \text{ in } P$:

1. *Constraint-based inference*: Determine τ and \mathcal{C} such that $\Gamma \vdash e : \tau \dashv \mathcal{C}$ is derivable

Recall: Putting everything together

input: program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \emptyset$

FOR EACH top-level let-expression $\text{let } x = e \text{ in } P$:

1. *Constraint-based inference*: Determine τ and \mathcal{C} such that $\Gamma \vdash e : \tau \dashv \mathcal{C}$ is derivable
2. *Unification*: Solve \mathcal{C} to get a most general unifier \mathcal{S} (**TYPE ERROR** if this fails)

Recall: Putting everything together

input: program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \emptyset$

FOR EACH top-level let-expression $\text{let } x = e \text{ in } P$:

1. *Constraint-based inference*: Determine τ and \mathcal{C} such that $\Gamma \vdash e : \tau \dashv \mathcal{C}$ is derivable
2. *Unification*: Solve \mathcal{C} to get a most general unifier \mathcal{S} (**TYPE ERROR** if this fails)
3. *Generalization*: Quantify over the free variables in $\mathcal{S}\tau$ to get the principle type $\forall \alpha_1 \dots \forall \alpha_k. \mathcal{S}\tau$ of e

Recall: Putting everything together

input: program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \emptyset$

FOR EACH top-level let-expression $\text{let } x = e \text{ in } P$:

1. *Constraint-based inference*: Determine τ and \mathcal{C} such that $\Gamma \vdash e : \tau \dashv \mathcal{C}$ is derivable
2. *Unification*: Solve \mathcal{C} to get a most general unifier \mathcal{S} (**TYPE ERROR** if this fails)
3. *Generalization*: Quantify over the free variables in $\mathcal{S}\tau$ to get the principle type $\forall \alpha_1 \dots \forall \alpha_k. \mathcal{S}\tau$ of e
4. Add $(x : \forall \alpha_1 \dots \forall \alpha_k. \mathcal{S}\tau)$ to Γ

Recall: As a Type System

$$\frac{}{\Gamma \vdash \epsilon} (\text{emptyProg})$$

$$\frac{\Gamma \vdash e : \tau \dashv \mathcal{C} \quad \Gamma, x : \text{principle}(\tau, \mathcal{C}) \vdash P}{\Gamma \vdash \text{let } x = e \ P} (\text{topLet})$$

We can also express this as a type system with judgments of the form $\Gamma \vdash P$, where P is a program (note there is no ":")

Recall: Example

```
let id = fun x -> x
let a = id (id 2 = 2)
```

Recall: Example

```
let id = fun x -> x
let a = id (id 2 = 2)
```

$\emptyset \vdash \text{let id} = \text{fun } x \rightarrow x \text{ let a} = \text{id } (\text{id } 2 = 2)$ (topLet)

Recall: Example

```
let id = fun x -> x
let a = id (id 2 = 2)
```

$\emptyset \vdash \text{let id} = \text{fun } x \rightarrow x \text{ let a} = \text{id } (\text{id } 2 = 2)$ (topLet)

$\vdash \emptyset \vdash \text{fun } x \rightarrow x : \alpha \rightarrow \alpha \dashv \emptyset$ (...)

$\vdash \{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha \} \vdash \text{let a} = \text{id } (\text{id } 2 = 2)$ (...)

Recall: Example

```
let id = fun x -> x
let a = id (id 2 = 2)
```

$\emptyset \vdash \text{let id} = \text{fun } x \rightarrow x \text{ let a} = \text{id } (\text{id } 2 = 2)$	(topLet)
$\vdash \emptyset \vdash \text{fun } x \rightarrow x : \alpha \rightarrow \alpha \dashv \emptyset$	(...)
$\vdash \emptyset \vdash \dots$	(...)
$\vdash \{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha \} \vdash \text{let a} = \text{id } (\text{id } 2 = 2)$	(topLet)
$\vdash \{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha \} \vdash \text{id } (\text{id } 2 = 2) : \alpha \dashv \mathcal{C}$	(...)
$\vdash \{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha \} \vdash \dots$	(...)
$\vdash \{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha , \text{ a} : \text{bool} \} \vdash \epsilon$	

Recall: Example

$\emptyset \vdash \text{let id} = \text{fun } x \rightarrow x \text{ let a} = \text{id} (\text{id} 2 = 2)$	(topLet)
$\vdash \emptyset \vdash \text{fun } x \rightarrow x : \alpha \rightarrow \alpha \dashv \emptyset$	(...)
$\vdash \emptyset \vdash \dots$	(...)
$\vdash \{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha \} \vdash \text{let a} = \text{id} (\text{id} 2 = 2)$	(topLet)
$\vdash \{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha \} \vdash \text{id} (\text{id} 2 = 2) : \alpha \dashv \mathcal{C}$	(...)
$\vdash \{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha \} \vdash \dots$	(...)
$\vdash \{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha , \text{ a} : \text{bool} \} \vdash \epsilon$	(emptyProg)

```
let id = fun x -> x
let a = id (id 2 = 2)
```

Practice Problem

Show that $\text{let } f = \lambda x. x \text{ in } f(f 2 = 2)$ has no principle type

Specialization

Recall: HM⁻ (Syntax)

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

Recall: HM^- (Typing)

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \text{int} \dashv \emptyset} \text{ (int)}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 = e_2 : \text{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \text{ (eq)}$$

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

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

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

Recall: HM⁻ (Typing Variables)

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

If x is declared in Γ , then x can be given the type τ with *all free variables replaced by **fresh variables***

This is where the polymorphism magic happens

Fresh variables can be unified with anything

An Alternative Formulation

$$\Gamma \vdash e : \tau$$

It's possible to give a type system for HM-
without constraints

It's very similar to the system from the first
half of the course, but with some rules for
dealing with **quantification** and **specialization**

HM⁻ (Alternative Typing)

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \text{int}} \text{ (int)}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{ (if)}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \text{bool}} \text{ (eq)}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ (add)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{ (fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \text{ (app)}$$

$$\frac{\tau_1 \text{ is a monotype} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{ (let)}$$

HM⁻ (Alternative Typing)

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \text{int}} \text{ (int)}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{ (if)}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \text{bool}} \text{ (eq)}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ (add)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{ (fun)}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \text{ (app)}$$

$$\frac{\tau_1 \text{ is a monotype} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{ (let)}$$

familiar rules

Generalization and Specialization

$$\frac{\Gamma \vdash e : \tau \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash e : \forall \alpha . \tau} \text{ (gen)} \quad \frac{(x : \tau) \in \Gamma \quad \tau \sqsubseteq \tau'}{\Gamma \vdash x : \tau'} \text{ (var)}$$

Generalization and Specialization

$$\frac{\Gamma \vdash e : \tau \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash e : \forall \alpha . \tau} \text{ (gen)} \quad \frac{(x : \tau) \in \Gamma \quad \tau \sqsubseteq \tau'}{\Gamma \vdash x : \tau'} \text{ (var)}$$

The generalization rule allows us to create polymorphic types

Generalization and Specialization

$$\frac{\Gamma \vdash e : \tau \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash e : \forall \alpha . \tau} \text{ (gen)} \quad \frac{(x : \tau) \in \Gamma \quad \tau \sqsubseteq \tau'}{\Gamma \vdash x : \tau'} \text{ (var)}$$

The generalization rule allows us to create polymorphic types

We also introduce a notion of **specialization** which allows us to *instance* polymorphic types at particular types

Generalization and Specialization

$$\frac{\Gamma \vdash e : \tau \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash e : \forall \alpha . \tau} \text{ (gen)} \quad \frac{(x : \tau) \in \Gamma \quad \tau \sqsubseteq \tau'}{\Gamma \vdash x : \tau'} \text{ (var)}$$

The generalization rule allows us to create polymorphic types

We also introduce a notion of **specialization** which allows us to *instance* polymorphic types at particular types

" \sqsubseteq " defined a *partial order* on type schemes

Specialization (Informal)

$$\forall \alpha_1 \dots \forall \alpha_m . \tau \sqsubseteq \forall \beta_1 \dots \forall \beta_n . \tau'$$

A type scheme T_2 **specializes** T_1 , written $T_1 \sqsubseteq T_2$ if T_2 the result of instantiating the bound variables of T_1 and generalizing over some of the variables introduced by the instantiation

Specialization (Formal)

τ_1, \dots, τ_m are monotypes

$$\tau' = [\tau_m/\alpha_m] \dots [\tau_1/\alpha_1] \tau$$

$$\beta_1, \dots, \beta_n \notin \text{FV}(\tau) \setminus \{\alpha_1, \dots, \alpha_m\}$$

$$\forall \alpha_1 \dots \forall \alpha_m. \tau \sqsubseteq \forall \beta_1 \dots \forall \beta_n. \tau'$$

A *specialization* of a type scheme is an instantiation of its bound variable, together with some generalizations over remaining free variables

Examples

Examples

$$\begin{aligned}\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha &\sqsubseteq \forall \eta. \eta \rightarrow \text{bool} \rightarrow \eta \\ &\sqsubseteq \text{int} \rightarrow \text{bool} \rightarrow \text{int}\end{aligned}$$

Examples

$$\begin{aligned}\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha &\sqsubseteq \forall \eta. \eta \rightarrow \text{bool} \rightarrow \eta \\ &\sqsubseteq \text{int} \rightarrow \text{bool} \rightarrow \text{int}\end{aligned}$$

$$\begin{aligned}\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha &\sqsubseteq \forall \gamma. \text{bool} \rightarrow (\gamma \rightarrow \gamma) \rightarrow \text{bool} \\ &\sqsubseteq \text{bool} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{bool}\end{aligned}$$

Examples

$$\begin{aligned}\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha &\sqsubseteq \forall \eta. \eta \rightarrow \text{bool} \rightarrow \eta \\ &\sqsubseteq \text{int} \rightarrow \text{bool} \rightarrow \text{int}\end{aligned}$$

$$\begin{aligned}\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha &\sqsubseteq \forall \gamma. \text{bool} \rightarrow (\gamma \rightarrow \gamma) \rightarrow \text{bool} \\ &\sqsubseteq \text{bool} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{bool}\end{aligned}$$

$$\begin{aligned}\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha &\sqsubseteq \text{bool} \rightarrow (\gamma \rightarrow \gamma) \rightarrow \text{bool} \\ &\not\sqsubseteq \text{bool} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{bool}\end{aligned}$$

Specialization and Principle Types

Specialization and Principle Types

Theorem. If $\Gamma \vdash e : \tau'$ then there is a type τ and constraints \mathcal{C} such that $\Gamma \vdash e : \tau \dashv \mathcal{C}$ and $\text{principle}(\tau, \mathcal{C}) \sqsubseteq \tau'$

Specialization and Principle Types

Theorem. If $\Gamma \vdash e : \tau'$ then there is a type τ and constraints \mathcal{C} such that $\Gamma \vdash e : \tau \dashv \mathcal{C}$ and $\text{principle}(\tau, \mathcal{C}) \sqsubseteq \tau'$

Theorem. If $\Gamma \vdash e : \tau \dashv \mathcal{C}$ and $\text{principle}(\tau, \mathcal{C}) \sqsubseteq \tau'$ then $\Gamma \vdash e : \tau'$

Specialization and Principle Types

Theorem. If $\Gamma \vdash e : \tau'$ then there is a type τ and constraints \mathcal{C} such that $\Gamma \vdash e : \tau \dashv \mathcal{C}$ and $\text{principle}(\tau, \mathcal{C}) \sqsubseteq \tau'$

Theorem. If $\Gamma \vdash e : \tau \dashv \mathcal{C}$ and $\text{principle}(\tau, \mathcal{C}) \sqsubseteq \tau'$ then $\Gamma \vdash e : \tau'$

The principle type is the most general "lowest" type with respect to specialization

Example

$$\{f : \forall \alpha. \alpha \rightarrow \alpha\} \vdash f(f\ 2 = 2) : \text{bool}$$

Why use constraints at all?

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

Why use constraints at all?

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

The alternative type rules are theoretically nice but not *algorithmic*

Why use constraints at all?

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

The alternative type rules are theoretically nice but not *algorithmic*

How do I choose which specialization to use in a derivation?

Why use constraints at all?

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

The alternative type rules are theoretically nice but not *algorithmic*

How do I choose which specialization to use in a derivation?

Constraints allow us to determine *which* specializations we should use *after the fact*

demo
(constraint-based inference)

HM⁻ (Typing Integers)

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \text{int} \dashv \emptyset} \text{ (int)}$$

Recall: HM^- (Typing Addition)

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

Recall: HM⁻ (Typing Equality)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 = e_2 : \text{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \text{ (eq)}$$

Recall II: HM⁻ (Typing If-Expressions)

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

HM⁻ (Typing Let-Expressions)

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

Recall: HM⁻ (Typing Functions)

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

Recall: HM^- (Typing Applications)

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

Recall: HM⁻ (Typing Variables)

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

Summary

The **principle type** of an expression is the most general type we could give it

Specialization defines a partial ordering on type schemes from most to least general

Our unification algorithm gives us a most general unifier, which will always give us the principle type of an expression