

Principle Types

Concepts of Programming Languages

Lecture 23

Outline

- » Demo an implementation of **unification**
- » Discuss **principle types** and **specialization**

Practice Problem

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

Determine the type τ and constraints \mathcal{C} such that the above judgment is derivable

Answer

$$\bullet \vdash \lambda x. x x : \overset{\tau}{\alpha} \rightarrow \overset{C}{\beta} \dashv \dot{\alpha} \doteq \alpha \rightarrow \beta$$

$$\lfloor \{ x : \alpha \} \vdash x x : \beta + \alpha \doteq \alpha \rightarrow \beta$$

$$\lfloor \{ x : \alpha \} \vdash x : \alpha \dashv \phi$$

$$\lfloor \{ x : \alpha \} \vdash x : \alpha \dashv \phi$$

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

Recap

Recall: Unification

$$\begin{array}{l} a \doteq d \rightarrow e \\ c \doteq \text{int} \rightarrow \boxed{d} \\ \text{int} \xrightarrow{\boxed{}} \boxed{\text{int} \rightarrow \boxed{\text{int}}} \doteq b \rightarrow \boxed{c} \end{array}$$

Unification is the process of solving a system of equations over *symbolic* expressions

It's kind of like solving a system of linear equations, but instead of working over real numbers and addition, we work over *uninterpreted* operators

Recall: ADT Unification Problem

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** (values of an ADT with variables)

Example: List Unification

```
type int_list =  
  | Nil  
  | Cons of int * int_list
```

$$x \doteq \text{Cons}(z, z')$$

$$z' \doteq \text{Cons}(z'', a)$$

$$a \doteq \text{Nil}$$

Example: Type Unification

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

Type unification is the unification problem for an ADT of types (with type variables acting as variables in the unification problem)

Recall: Unifiers

A **unifier** is a sequence of substitutions to variables, typically written

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

We write $\mathcal{S}t$ for $[t_n/x_n] \dots [t_1/x_1]t$. A solution must have the property that it **satisfies** every equation

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

Recall: Most General Unifiers

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

In other words, \mathcal{S}' is \mathcal{S} *with more substitutions*

4

$$\alpha \doteq \alpha$$

$$\mathcal{S}_1 = \{ \alpha \mapsto \text{int} \}$$

$$\mathcal{S}_2 = \{ \alpha \mapsto \text{bool} \}$$

$$\mathcal{S}_3 = \{ \alpha \mapsto \beta \}$$

MGU:

$$\mathcal{S} = \emptyset$$

An Algorithm (Pseudocode)

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

MATCH eq :

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

MATCH eq :

$t_1 \doteq t_2$ where $t_1 = t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$ *// if t_1 and t_2 are syntactically equal then remove eq*

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

MATCH eq :

$t_1 \doteq t_2$ where $t_1 = t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$ *// if t_1 and t_2 are syntactically equal then remove eq*

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// remove eq and add $s_1 \doteq s_2$ and $t_1 \doteq t_2$*

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

MATCH eq :

$t_1 \doteq t_2$ where $t_1 = t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$ *// if t_1 and t_2 are syntactically equal then remove eq*

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// remove eq and add $s_1 \doteq s_2$ and $t_1 \doteq t_2$*

$\alpha \doteq t$ or $t \doteq \alpha$ where $\alpha \notin \text{FV}(t) \implies$ *// type variable α does not appear free in t*

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

MATCH eq :

$t_1 \doteq t_2$ where $t_1 = t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$ *// if t_1 and t_2 are syntactically equal then remove eq*

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// remove eq and add $s_1 \doteq s_2$ and $t_1 \doteq t_2$*

$\alpha \doteq t$ or $t \doteq \alpha$ where $\alpha \notin \text{FV}(t) \implies$ *// type variable α does not appear free in t*

$\mathcal{S} \leftarrow \mathcal{S} \cup \{\alpha \mapsto t\}$ *// add $\alpha \mapsto t$ to \mathcal{S}*

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

MATCH eq :

$t_1 \doteq t_2$ where $t_1 = t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$ *// if t_1 and t_2 are syntactically equal then remove eq*

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// remove eq and add $s_1 \doteq s_2$ and $t_1 \doteq t_2$*

$\alpha \doteq t$ or $t \doteq \alpha$ where $\alpha \notin \text{FV}(t) \implies$ *// type variable α does not appear free in t*

$\mathcal{S} \leftarrow \mathcal{S} \cup \{\alpha \mapsto t\}$ *// add $\alpha \mapsto t$ to \mathcal{S}*

$\mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

MATCH eq :

$t_1 \doteq t_2$ where $t_1 = t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$ *// if t_1 and t_2 are syntactically equal then remove eq*

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// remove eq and add $s_1 \doteq s_2$ and $t_1 \doteq t_2$*

$\alpha \doteq t$ or $t \doteq \alpha$ where $\alpha \notin \text{FV}(t) \implies$ *// type variable α does not appear free in t*

$\mathcal{S} \leftarrow \mathcal{S} \cup \{\alpha \mapsto t\}$ *// add $\alpha \mapsto t$ to \mathcal{S}*

$\mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$

perform the substitution $\alpha \mapsto t$ to every equation in \mathcal{U}

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

MATCH eq :

$t_1 \doteq t_2$ where $t_1 = t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$ *// if t_1 and t_2 are syntactically equal then remove eq*

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// remove eq and add $s_1 \doteq s_2$ and $t_1 \doteq t_2$*

$\alpha \doteq t$ or $t \doteq \alpha$ where $\alpha \notin \text{FV}(t) \implies$ *// type variable α does not appear free in t*

$\mathcal{S} \leftarrow \mathcal{S} \cup \{\alpha \mapsto t\}$ *// add $\alpha \mapsto t$ to \mathcal{S}*

$\mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$

perform the substitution $\alpha \mapsto t$ to every equation in \mathcal{U}

OTHERWISE \implies **FAIL**

An Algorithm (Pseudocode)

input: type unification problem \mathcal{U}

output: most general unifier to \mathcal{U}

$\mathcal{S} \leftarrow$ empty solution

WHILE $eq \in \mathcal{U}$: *// \mathcal{U} is not empty*

MATCH eq :

$t_1 \doteq t_2$ where $t_1 = t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$ *// if t_1 and t_2 are syntactically equal then remove eq*

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// remove eq and add $s_1 \doteq s_2$ and $t_1 \doteq t_2$*

$\alpha \doteq t$ or $t \doteq \alpha$ where $\alpha \notin \text{FV}(t) \implies$ *// type variable α does not appear free in t*

$\mathcal{S} \leftarrow \mathcal{S} \cup \{\alpha \mapsto t\}$ *// add $\alpha \mapsto t$ to \mathcal{S}*

$\mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\}$

perform the substitution $\alpha \mapsto t$ to every equation in \mathcal{U}

OTHERWISE \implies **FAIL**

RETURN \mathcal{S}

$$U = \{ \cancel{\alpha \vdash \alpha \rightarrow \beta} \}$$

$$S = \{ \alpha \mapsto \alpha \rightarrow \beta \}$$

not valid because
 $\alpha \vdash FV(\alpha \rightarrow \beta)$

$$S\alpha = \alpha \rightarrow \beta \neq (\alpha \rightarrow \beta) \rightarrow \beta = S(\alpha \rightarrow \beta)$$

$$(\dots (a \dots \rightarrow \beta \rightarrow \beta) \rightarrow \beta$$

Example

$$\cancel{a \doteq d \rightarrow e}$$

$$\text{var} \doteq \text{ty}$$

$$\cancel{c \doteq \text{int} \rightarrow d}$$

$$\text{var} \doteq \text{ty}$$

$$\cancel{\text{int} \rightarrow \text{int} \rightarrow \text{int} \doteq b \rightarrow \cancel{e} \mid \text{int} \rightarrow d}$$

$$\text{fun ty} \doteq \text{fun ty}$$

$$\cancel{\text{int} \doteq b}$$

$$\text{ty} \doteq \text{var}$$

$$\cancel{\text{int} \rightarrow \text{int} \doteq \text{int} \rightarrow d}$$

$$\text{fun ty} \doteq \text{fun ty}$$

$$\cancel{\text{int} \doteq \text{int}}$$

$$\text{eq}$$

$$\cancel{\text{int} \doteq d}$$

$$\text{ty} \doteq \text{var}$$

$$S = \{ a \mapsto d \rightarrow e$$

$$c \mapsto \text{int} \rightarrow d$$

$$b \mapsto \text{int}$$

$$d \mapsto \text{int} \}$$

Another Practice Problem

$$\beta \doteq \eta$$

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

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

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

Determine a most general unifier to the above type unification problem using the algorithm we just gave

Answer

$$\begin{aligned}
 & \beta \doteq \eta \quad \text{var} \doteq \text{ty} \\
 & \alpha \xrightarrow{\eta} \beta \doteq \alpha \rightarrow \gamma \quad \text{fn} \doteq \text{fn} \\
 & \alpha \xrightarrow{\eta} \beta \doteq \gamma \Rightarrow \eta \quad \text{fn} \doteq \text{fn} \\
 & \alpha \xrightarrow{\eta} \beta \doteq \text{int} \rightarrow \eta \quad \text{fn} \doteq \text{fn} \\
 & \alpha \doteq \alpha \quad \text{eq} \quad \text{int} \quad \text{int} \\
 & \eta \doteq \gamma \quad \text{v} \doteq \text{t} \\
 & \alpha \doteq \gamma \quad \text{v} \doteq \text{t} \\
 & \gamma \eta \doteq \gamma \gamma \quad \text{eq} \\
 & \gamma \alpha \doteq \text{int} \quad \text{v} \doteq \text{t}
 \end{aligned}$$

eg

$$S = \left\{ \begin{array}{l} \beta \mapsto \eta \\ \eta \mapsto \gamma \\ \alpha \mapsto \gamma \\ \gamma \mapsto \text{int} \end{array} \right\}$$

another MGU;

$$S = \left\{ \begin{array}{l} \beta \mapsto \text{int} \\ \eta \mapsto \text{int} \\ \gamma \mapsto \text{int} \\ \alpha \mapsto \text{int} \end{array} \right\}$$

demo
(unification)

Principle Types

Principle Types

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

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 :

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 :

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

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 $\forall \alpha_1, \dots, \alpha_k. \mathcal{S}\tau$

Example

Determine the principle type of $\lambda f. \lambda x. f(x + 1)$

Example

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

Putting everything together

Putting everything together

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

Putting everything together

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

$\Gamma \leftarrow \emptyset$

Putting everything together

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

$\Gamma \leftarrow \emptyset$

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

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

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)

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

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

Example

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

$$\Gamma = \emptyset$$

$$\downarrow \text{let } id = \text{fun } x \rightarrow x$$

$$\Gamma = \{ id : \forall \alpha. \alpha \rightarrow \alpha \}$$

$$\downarrow \text{let } _ = id (id \ 2 = 2)$$

$$\Gamma = \{ id : \forall \alpha. \alpha \rightarrow \alpha, _ : bool \}$$

Specialization

Recall: HM⁻ (Syntax)

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

Recall: HM⁻ (Typing)

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

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

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

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

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

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

Recall: HM⁻ (Typing Variables)

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

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 our 320Caml system, but
with some rules for dealing with **quantification**
and **specialization**

HM⁻ (Alternative Typing)

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \text{int}} \quad (\text{int}) \qquad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (\text{if})$$

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

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

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

HM⁻ (Alternative Typing)

familiar rules

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \text{int}} \quad (\text{int}) \qquad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (\text{if})$$

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

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

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

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 is like the one from System F

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 is like the one from System F

The main difference: we introduce a notion of **specialization** which allows us to *instantiate* polymorphic functions 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 is like the one from System F

The main difference: we introduce a notion of **specialization** which allows us to *instantiate* polymorphic functions 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?

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

Why use constraints at all?

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

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

Why use constraints at all?

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

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?

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

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*

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