Principle Types

Concepts of Programming Languages
Lecture 23

CAS CS 320

Outline

» Demo an implementation of **unification**

» Discuss **principle types** and **specialization**

Practice Problem

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

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

Answer

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

Recap

Recall: Unification

$$
\begin{aligned}
a &\doteq d \to e \\
c &\doteq \mathsf{int} \to d \\
\mathsf{int} \to \mathsf{int} \to \mathsf{int} &\doteq b \to c
\end{aligned}
$$

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

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

where $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$ are **terms** (values of an ADT with variables)

Example: List Unification

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

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

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

We write $\mathcal{S}t$ for $[t_n/x_n]\ldots[t_1/x_1]t$. 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*

An Algorithm (Pseudocode)

An Algorithm (Pseudocode)

```
input: type unification problem 𝒰
output: most general unifier to 𝒰
```

An Algorithm (Pseudocode)

<u>input:</u> type unification problem $\mathscr{U}$
<u>output:</u> most general unifier to $\mathscr{U}$

$\mathscr{S} \leftarrow$ empty solution

An Algorithm (Pseudocode)

```
input: type unification problem 𝒰
output: most general unifier to 𝒰

𝒮 ← empty solution

WHILE eq ∈ 𝒰:  // 𝒰 is not empty
```

An Algorithm (Pseudocode)

```
input: type unification problem 𝒰
output: most general unifier to 𝒰

𝒮 ← empty solution

WHILE eq ∈ 𝒰:  // 𝒰 is not empty

  MATCH eq:
```

An Algorithm (Pseudocode)

input: type unification problem $\mathscr{U}$
output: most general unifier to $\mathscr{U}$

$\mathscr{S} \leftarrow$ empty solution

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

  MATCH eq:

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

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

An Algorithm (Pseudocode)

<u>input:</u> type unification problem $\mathcal{U}$
<u>output:</u> 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 $t_1 = t_2$ $\Longrightarrow$ $\mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\}$ // if t_1 and t_2 are *syntactically* equal then remove eq

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

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

An Algorithm (Pseudocode)

```
input: type unification problem 𝒰
output: most general unifier to 𝒰

𝒮 ← empty solution

WHILE eq ∈ 𝒰:   // 𝒰 is not empty

  MATCH eq:

    t₁ ≐ t₂ where t₁ = t₂ ⟹ 𝒰 ← 𝒰\{eq}   // if t₁ and t₂ are syntactically equal then remove eq

    s₁ → t₁ ≐ s₂ → t₂ ⟹ 𝒰 ← 𝒰\{eq} ∪ {s₁ ≐ s₂, t₁ ≐ t₂} // remove eq and add s₁ ≐ s₂ and t₁ ≐ t₂

    α ≐ t or t ≐ α where α ∉ FV(t) ⟹ // type variable α does not appear free in t

      𝒮 ← 𝒮 ∪ {α ↦ t}   // add α ↦ t to 𝒮
```

An Algorithm (Pseudocode)

<u>input:</u> `type unification problem` $\mathcal{U}$
<u>output:</u> `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` $t_1 = t_2$ $\Longrightarrow$ $\mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\}$ `// if` t_1 `and` t_2 `are` *`syntactically`* `equal then remove` eq

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2$ $\Longrightarrow$ $\mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ `// remove` eq `and add` $s_1 \doteq s_2$ `and` $t_1 \doteq t_2$

$\alpha \doteq t$ `or` $t \doteq \alpha$ `where` $\alpha \notin \mathrm{FV}(t)$ $\Longrightarrow$ `// type variable` α `does not appear free in` t

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

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

An Algorithm (Pseudocode)

```
input: type unification problem 𝒰
output: most general unifier to 𝒰
```

$\mathcal{S} \leftarrow$ empty solution

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

MATCH eq:

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

$s_1 \to t_1 \doteq s_2 \to t_2$ $\Longrightarrow$ $\mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ // remove eq and add $s_1 \doteq s_2$ and $t_1 \doteq t_2$

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

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

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

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

An Algorithm (Pseudocode)

<u>input:</u> type unification problem $\mathcal{U}$
<u>output:</u> 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 $t_1 = t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\}$ // if t_1 and t_2 are *syntactically* equal then remove eq

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

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

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

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

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

OTHERWISE $\implies$ **FAIL**

An Algorithm (Pseudocode)

```
input: type unification problem 𝒰
output: most general unifier to 𝒰

𝒮 ← empty solution

WHILE eq ∈ 𝒰:   // 𝒰 is not empty

  MATCH eq:

    t₁ ≐ t₂ where t₁ = t₂ ⟹ 𝒰 ← 𝒰\{eq}   // if t₁ and t₂ are syntactically equal then remove eq

    s₁ → t₁ ≐ s₂ → t₂ ⟹ 𝒰 ← 𝒰\{eq} ∪ {s₁ ≐ s₂, t₁ ≐ t₂} // remove eq and add s₁ ≐ s₂ and t₁ ≐ t₂

    α ≐ t or t ≐ α where α ∉ FV(t) ⟹ // type variable α does not appear free in t

      𝒮 ← 𝒮 ∪ {α ↦ t}   // add α ↦ t to 𝒮

      𝒰 ← 𝒰\{eq}

      perform the substitution α ↦ t to every equation in 𝒰

    OTHERWISE ⟹ FAIL

RETURN 𝒮
```

Example

$$a \doteq d \to e$$
$$c \doteq \mathsf{int} \to d$$
$$\mathsf{int} \to \mathsf{int} \to \mathsf{int} \doteq b \to c$$

Another Practice Problem

$$\beta \doteq \eta$$

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

$$\alpha \to \beta \doteq \gamma \to \eta$$

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

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

Answer

$$\beta \doteq \eta$$

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

$$\alpha \to \beta \doteq \gamma \to \eta$$

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

demo

(unification)

Principle Types

Principle Types

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

Principle Types

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

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

Principle Types

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

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

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

Principle Types

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

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

$$\text{principle}(\tau, \mathscr{C}) = \forall \alpha_1 \ldots \forall \alpha_k . \mathscr{S}\tau \text{ where } \mathsf{FV}(\mathscr{S}\tau) = \{\alpha_1, \ldots, \alpha_k\}$$

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

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

<u>input:</u> program P (sequence of top-level let-expressions)

Putting everything together

<u>input:</u> program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \varnothing$

Putting everything together

<u>input:</u> program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \varnothing$

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

Putting everything together

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

$\Gamma \leftarrow \varnothing$

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

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

Putting everything together

<u>input:</u> program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \varnothing$

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

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

2. *Unification:* Solve $\mathscr{C}$ to get a most general unifier $\mathscr{S}$ (**TYPE ERROR** if this fails)

Putting everything together

<u>input:</u> program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \varnothing$

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

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

2. *Unification:* Solve $\mathscr{C}$ to get a most general unifier $\mathscr{S}$ (**TYPE ERROR** if this fails)

3. *Generalization:* Quantify over the free variables in $\mathscr{S}\tau$ to get the principle type $\forall\alpha_1\ldots\forall\alpha_k.\mathscr{S}\tau$ of e

Putting everything together

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

$\Gamma \leftarrow \varnothing$

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

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

Example

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

Specialization

Recall: HM^{-} (Syntax)

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

Recall: HM$^-$ (Typing)

$$\frac{n \texttt{ is an integer}}{\Gamma \vdash n : \mathsf{int} \dashv \varnothing}\ \texttt{(int)}$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathscr{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathscr{C}_2}{\Gamma \vdash e_1 = e_2 : \mathsf{bool} \dashv \tau_1 \doteq \tau_2, \mathscr{C}_1, \mathscr{C}_2}\ \texttt{(eq)}$$

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

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathscr{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathscr{C}_2 \qquad \alpha \texttt{ is fresh}}{\Gamma \vdash e_1 e_2 : \alpha \dashv \tau_1 \doteq \tau_2 \to \alpha, \mathscr{C}_1, \mathscr{C}_2}\ \texttt{(app)}$$

Recall: HM$^-$ (Typing Variables)

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

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)

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

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \dashv \mathscr{C}}{\Gamma \vdash \lambda x\,.\,e : \tau_1 \to \tau_2 \dashv \mathscr{C}} \ (\texttt{fun}) \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \ (\texttt{app})$$

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

HM$^-$ (Alternative Typing)

familiar rules

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

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}\ (\texttt{if})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \mathsf{bool}}\ (\texttt{eq})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}\ (\texttt{add})$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \dashv \mathscr{C}}{\Gamma \vdash \lambda x \,.\, e : \tau_1 \to \tau_2 \dashv \mathscr{C}}\ (\texttt{fun})$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}\ (\texttt{app})$$

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

Generalization and Specialization

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

Generalization and Specialization

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

The generalization rule is like the one from System F

Generalization and Specialization

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

The generalization rule is like the one from System F

<u>The main difference:</u> we introduce a notion of **specialization** which allows us to *instantiate* polymorphic functions at particular types

Generalization and Specialization

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

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)

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

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)

$$\frac{\tau_1, \ldots, \tau_m \text{ are monotypes} \quad \tau' = [\tau_m/\alpha_m]\ldots[\tau_1/\alpha_1]\tau \quad \beta_1, \ldots, \beta_n \notin \mathsf{FV}(\tau) \setminus \{\alpha_1, \ldots, \alpha_m\}}{\forall \alpha_1 \ldots \forall \alpha_m . \tau \sqsubseteq \forall \beta_1 \ldots \forall \beta_n . \tau'}$$

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

Examples

Examples

$$\begin{aligned}\forall\alpha\,.\,\forall\beta\,.\,\alpha \to \beta \to \alpha &\sqsubseteq \forall\eta\,.\,\eta \to \mathsf{bool} \to \eta \\ &\sqsubseteq \mathsf{int} \to \mathsf{bool} \to \mathsf{int}\end{aligned}$$

Examples

$$\begin{aligned}\forall\alpha\,.\,\forall\beta\,.\,\alpha \to \beta \to \alpha &\sqsubseteq \forall\eta\,.\,\eta \to \mathsf{bool} \to \eta \\ &\sqsubseteq \mathsf{int} \to \mathsf{bool} \to \mathsf{int}\end{aligned}$$

$$\begin{aligned}\forall\alpha\,.\,\forall\beta\,.\,\alpha \to \beta \to \alpha &\sqsubseteq \forall\gamma\,.\,\mathsf{bool} \to (\gamma \to \gamma) \to \mathsf{bool} \\ &\sqsubseteq \mathsf{bool} \to (\mathsf{int} \to \mathsf{int}) \to \mathsf{bool}\end{aligned}$$

Examples

$$\begin{aligned}\forall\alpha\,.\,\forall\beta\,.\,\alpha\to\beta\to\alpha &\sqsubseteq \forall\eta\,.\,\eta\to\mathsf{bool}\to\eta\\ &\sqsubseteq \mathsf{int}\to\mathsf{bool}\to\mathsf{int}\end{aligned}$$

$$\begin{aligned}\forall\alpha\,.\,\forall\beta\,.\,\alpha\to\beta\to\alpha &\sqsubseteq \forall\gamma\,.\,\mathsf{bool}\to(\gamma\to\gamma)\to\mathsf{bool}\\ &\sqsubseteq \mathsf{bool}\to(\mathsf{int}\to\mathsf{int})\to\mathsf{bool}\end{aligned}$$

$$\begin{aligned}\forall\alpha\,.\,\forall\beta\,.\,\alpha\to\beta\to\alpha &\sqsubseteq \mathsf{bool}\to(\gamma\to\gamma)\to\mathsf{bool}\\ &\not\sqsubseteq \mathsf{bool}\to(\mathsf{int}\to\mathsf{int})\to\mathsf{bool}\end{aligned}$$

Specialization and Principle Types

Specialization and Principle Types

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

Specialization and Principle Types

<u>Theorem.</u> If $\Gamma \vdash e : \tau'$ then there is a type τ and constraints $\mathscr{C}$ such that $\Gamma \vdash e : \tau \dashv \mathscr{C}$ and $\mathsf{principle}(\tau, \mathscr{C}) \sqsubseteq \tau'$

<u>Theorem.</u> If $\Gamma \vdash e : \tau \dashv \mathscr{C}$ and $\mathsf{principle}(\tau, \mathscr{C}) \sqsubseteq \tau'$ then $\Gamma \vdash e : \tau'$

Specialization and Principle Types

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

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

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

Example

$$\{f : \forall \alpha \,.\, \alpha \to \alpha\} \vdash f\ (f\ 2 = 2) : \mathsf{bool}$$

Why use constraints at all?

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

Why use constraints at all?

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

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

Why use constraints at all?

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

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?

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

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