Principle Types

Concepts of Programming Languages

CAS CS 320

Outline

- Demo an implementation of **unification**
- Discuss **principle types**

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

$$\frac{(x : \text{[redacted]}\ \tau) \in \Gamma \qquad \text{[redacted]}}{\Gamma \vdash x : \text{[redacted]}\ \tau \dashv \varnothing}\ (\texttt{var})$$

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

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

$\{\tau_1 \doteq \tau_2 \to \alpha\} \cup \mathscr{C}_1 \cup \mathscr{C}_2$

Answer

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

$\emptyset \vdash \lambda x . x x : \alpha \to \beta \dashv \alpha \doteq \alpha \to \beta$ (fun)

└ $\{x : \alpha\} \vdash x x : \beta \dashv \alpha \doteq \alpha \to \beta$ (app)

  ├ $\{x : \alpha\} \vdash x : \alpha \dashv \emptyset$ (var)

  └ $\{x : \alpha\} \vdash x : \alpha \dashv \emptyset$ (var)

$\cancel{\alpha \doteq (\alpha \to \beta) \to \beta}$

$\cancel{\alpha \doteq ((\alpha \to \beta) \to \beta) \to \beta}$

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

Recall: Type 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 **types**

Recall: Unifiers

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

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

s.t.

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

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)

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

An Algorithm (Pseudocode)

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

𝒮 ← 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 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
```

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
```

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

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

$s_1 \to t_1 \doteq s_2 \to t_2$ $\Longrightarrow$ $\mathscr{U} \leftarrow \mathscr{U} \backslash \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ // 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)

```
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}
```

An Algorithm (Pseudocode)

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

```
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
```

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
```

An Algorithm (Pseudocode)

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

```
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
```

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 $\mathscr{S}$ we can get the "actual" type of e:

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

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:

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

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 (is_well_typed)

Putting everything together (is_well_typed)

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

Putting everything together (is_well_typed)

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

$\Gamma \leftarrow \varnothing$

Putting everything together (is_well_typed)

input: 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 (is_well_typed)

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 $\mathcal{C}$ such that $\Gamma \vdash e : \tau \dashv \mathcal{C}$ is derivable

Putting everything together (is_well_typed)

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)

Putting everything together (is_well_typed)

<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 $\mathcal{S}$ (**TYPE ERROR** if this fails)
3. *Generalization:* Quantify over the free variables in $\mathcal{S}\tau$ to get the principle type $\forall \alpha_1 \ldots \forall \alpha_k . \mathcal{S}\tau$ of e

Putting everything together (is_well_typed)

<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
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)
```

As a Type System

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

$$\frac{\Gamma \vdash e : \tau \dashv \mathscr{C} \qquad \tau' = \text{principle}(\tau, \mathscr{C}) \qquad \Gamma, x : \tau' \vdash P}{\Gamma \vdash \text{let } x = e\ P}\ \texttt{(topLet)}$$

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 ":")

Example

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

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

Example

$\varnothing \vdash$ `let id = fun x -> x let a = id (id 2 = 2)` (topLet)

Example

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

$\varnothing \vdash$ `let id = fun x -> x let a = id (id 2 = 2)` (topLet)

├─ $\varnothing \vdash$ `fun x -> x` $: \alpha \to \alpha \dashv \varnothing$ (...)

│ | ... (...)

└─ $\{$`id` $: \forall \alpha . \alpha \to \alpha\} \vdash$ `let a = id (id 2 = 2)`

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

Example

$\varnothing \vdash$ `let id = fun x -> x let a = id (id 2 = 2)` (topLet)

- $\varnothing \vdash$ `fun x -> x` $: \alpha \to \alpha \dashv \varnothing$ (...)
  - ... (...)
- $\{$`id` $: \forall \alpha . \alpha \to \alpha\} \vdash$ `let a = id (id 2 = 2)` (topLet)
  - $\{$`id` $: \forall \alpha . \alpha \to \alpha\} \vdash$ `id (id 2 = 2)` $: \alpha \dashv \mathscr{C}$ (...)
    - ... (...)
  - $\{$`id` $: \forall \alpha . \alpha \to \alpha$, `a` $:$ `bool`$\} \vdash \epsilon$

Example

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

- $\varnothing \vdash$ `let id = fun x -> x let a = id (id 2 = 2)` (topLet)
  - $\varnothing \vdash$ `fun x -> x` $: \alpha \to \alpha \dashv \varnothing$ (...)
    - ... (...)
  - $\{$`id` $: \forall \alpha . \alpha \to \alpha\} \vdash$ `let a = id (id 2 = 2)` (topLet)
    - $\{$`id` $: \forall \alpha . \alpha \to \alpha\} \vdash$ `id (id 2 = 2)` $: \alpha \dashv \mathscr{C}$ (...)
      - ... (...)
    - $\{$`id` $: \forall \alpha . \alpha \to \alpha$, `a` $:$ `bool`$\} \vdash \epsilon$ (emptyProg)

Summary

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

Our unification algorithm gives us a **most general unifier**, which we use to get the principle type of an expression