

Principle Types

Concepts of Programming Languages

Outline

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

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

$$\frac{(x : \forall \alpha_1. \forall \alpha_2. \dots \forall \alpha_k. \tau) \in \Gamma \quad \beta_1, \dots, \beta_k \text{ are fresh}}{\Gamma \vdash x : \beta_1 \rightarrow \dots \rightarrow \beta_k \rightarrow \tau \dashv \emptyset} \text{ (var)} \qquad \frac{\alpha \text{ is fresh} \quad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \lambda x . e : \alpha \rightarrow \tau \dashv \mathcal{C}} \text{ (fun)}$$

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

Answer

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

$$\emptyset \vdash \lambda x. xx : \alpha \rightarrow \beta \dashv \alpha \doteq \alpha \rightarrow \beta$$

$$\vdash \{x : \alpha\} \vdash xx : \beta \dashv \alpha \doteq \alpha \rightarrow \beta$$

$$\left[\begin{array}{l} \vdash \{x : \alpha\} \vdash x : \alpha \dashv \emptyset \\ \vdash \{x : \alpha\} \vdash x : \alpha \dashv \emptyset \end{array} \right.$$

Recap

Recall: Unification

$$\begin{aligned}a &\doteq d \rightarrow e \\c &\doteq \text{int} \rightarrow d \\ \text{int} \rightarrow \text{int} \rightarrow \text{int} &\doteq b \rightarrow c\end{aligned}$$

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

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


where s_1, \dots, s_k and t_1, \dots, t_k are **types**

Recall: Unifiers

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

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

s.t.

 *ordered*

$$\mathcal{S}t_1 = \mathcal{S}s_1$$

$$\mathcal{S}s_2 = \mathcal{S}t_2$$

$$\vdots$$

$$\mathcal{S}s_k = \mathcal{S}t_k$$

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 \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}

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

$\beta \doteq \eta$ (asn)
 ~~$\alpha \rightarrow \beta \doteq \alpha \Rightarrow \gamma$~~ (fun)
 ~~$\alpha \rightarrow \beta \doteq \gamma \rightarrow \eta$~~ (fun)
 ~~$\alpha \rightarrow \beta \doteq \text{int} \rightarrow \eta$~~ (fun)
 ~~$\alpha \doteq \alpha$~~ (eq)
 ~~$\eta \doteq \gamma$~~ (asn)
 ~~$\alpha \doteq \gamma$~~ (asn)
 ~~$\gamma \rightarrow \eta \doteq \gamma$~~ (eq)
 ~~$\gamma \rightarrow \text{int}$~~ (asn)
 ~~$\eta \doteq \gamma \rightarrow \text{int}$~~ (eq)

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

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

$$S(\alpha \Rightarrow \beta \Rightarrow \eta) = (\text{int} \Rightarrow \boxed{\eta}) \rightarrow \text{int} \Rightarrow \boxed{\eta}$$

FV FV

$$\text{Prin}(\alpha \Rightarrow \beta \Rightarrow \eta, \mathcal{C}) = \forall \eta. (\text{int} \Rightarrow \eta) \rightarrow \text{int} \Rightarrow \eta$$

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

$$\vdash \lambda f. \lambda x. f(x+1) : \alpha \Rightarrow \beta \Rightarrow \eta \vdash \mathcal{C}$$

$$\vdash \{f : \alpha\} \vdash \lambda x. f(x+1) : \beta \Rightarrow \eta \vdash \mathcal{C}$$

$$\vdash \{f : \alpha, x : \beta\} \vdash f(x+1) : \eta \vdash \left[\begin{array}{l} \alpha \doteq \text{int} \rightarrow \eta \\ \beta \doteq \text{int} \\ \text{int} \doteq \text{int} \end{array} \right] \mathcal{C}$$

$$\vdash \{f : \alpha, x : \beta\} \vdash f : \alpha \vdash \emptyset$$

$$\vdash \{f : \alpha, x : \beta\} \vdash x+1 : \text{int} \vdash \beta \doteq \text{int}, \text{int} \doteq \text{int}$$

$$\vdash \{ \dots \} \vdash x : \beta \vdash \emptyset$$

$$\vdash \{ \dots \} \vdash 1 : \text{int} \vdash \emptyset$$

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

$$\beta \doteq \text{int}$$

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

$$S = \{$$

$$\alpha \mapsto \text{int} \rightarrow \eta$$

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

}

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)

$\Gamma \leftarrow \emptyset$

Putting everything together (is_well_typed)

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 (is_well_typed)

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 (is_well_typed)

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 (is_well_typed)

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 (is_well_typed)

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

$\Gamma = \emptyset$

let id = fun x -> x

let _ = id (id 2 = 2)

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



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

$_ : \text{bool}$

$\}$

$\emptyset \vdash \text{fun } x \rightarrow x : \tau \rightarrow \mathcal{C}$

$\text{prim}(\tau, \mathcal{C}) = \forall \alpha. \alpha \rightarrow \alpha$

$\{ \text{id} : \forall \alpha. \alpha \rightarrow \alpha \} \vdash \text{id } (\text{id } 2 = 2) :$
 $\tau \rightarrow \mathcal{C}$

$\text{prim}(\tau, \mathcal{C}) = \text{bool}$

As a Type System

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

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

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 ":")

Example

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

Example

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

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

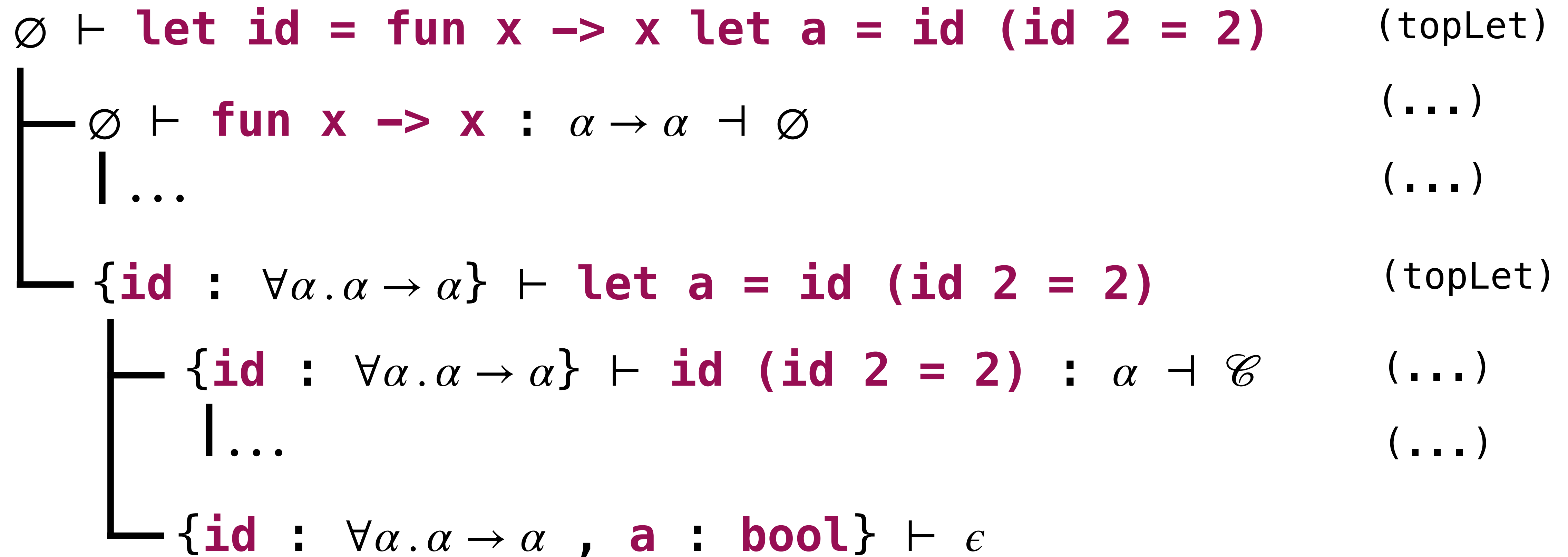
Example

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

$\emptyset \vdash$	let id = fun x -> x let a = id (id 2 = 2)	(topLet)
\vdash	$\emptyset \vdash$	(...)
\vdash	fun x -> x : $\alpha \rightarrow \alpha \dashv \emptyset$	(...)
\vdash	...	
\vdash	{id : $\forall \alpha. \alpha \rightarrow \alpha$} \vdash let a = id (id 2 = 2)	

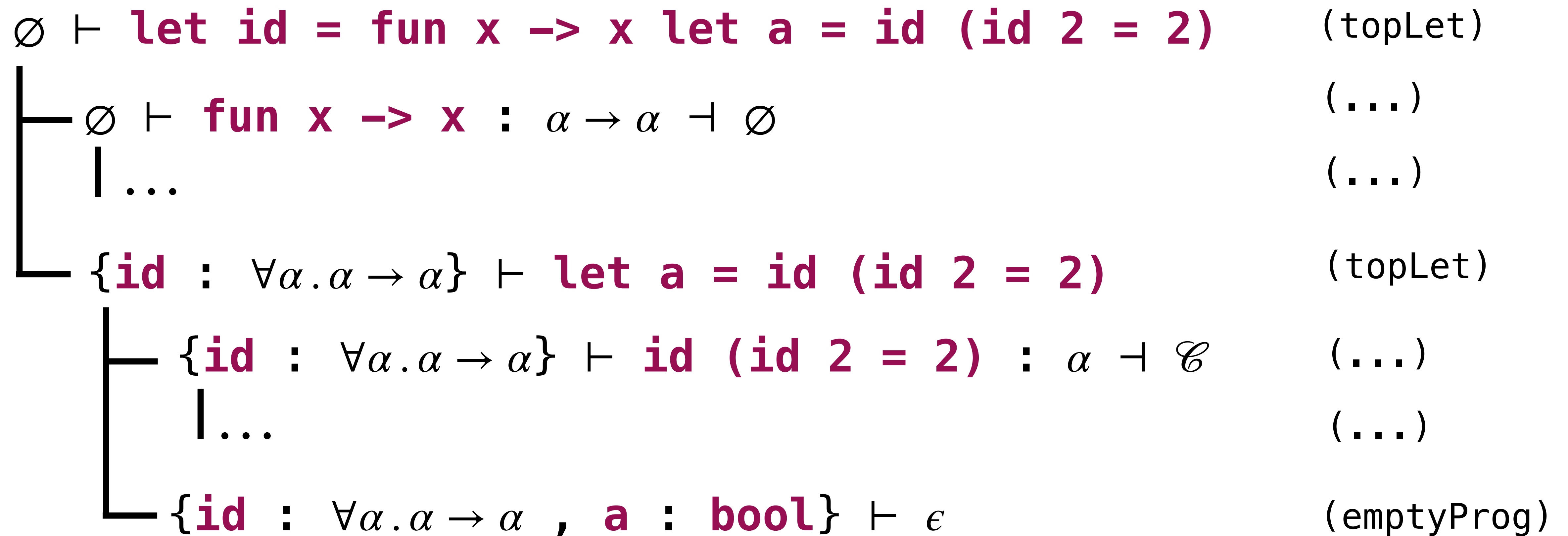
Example

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



Example

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



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