Unification

Concepts of Programming Languages
Lecture 22

CAS CS 320

Outline

» Finish up our discussion of **Hindley–Milner Light** (HM$^-$)

» Briefly discuss **let-polymorphism**

» Describe the **unification** algorithm used to determine the "actual" type of our expression, given a collection of constraints

Recap

Recall: Parametric Polymorphism

```
let rec rev = function
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

Parametric polymorphism allows for functions which are agnostic to the types of its inputs

For example, we can write a single reverse function and use it in multiple contexts

Recall: Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

`In reality, types variables in OCaml are` **`quantified`**

`We read this "`**`id`**` has type `**`t -> t`**` for any type `**`t`**`"`

Recall: Hindley-Milner Light

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

Recall: Type Schemes

$$\sigma ::= \mathsf{int} \mid \mathsf{bool} \mid \alpha \mid \sigma \to \sigma$$
$$\tau ::= \sigma \mid \forall \alpha \,.\, \tau$$

σ represents **monotypes**, types with *no quantification*. A type is **monomorphic** if it is a monotype with no type variables

τ represents **type schemes**, which are types with some number of quantified type variables

We say a type is **polymorphic** if it is a *closed* type scheme

Recall: Constraint-Based Inference

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

Our typing rules well need to keep track of a set of **constraints**, which tell use what must hold for e to be well-typed

The idea: We're formalizing the idea of "collecting together" our constraints, as in our intuitive example

Recall: What is a constraint?

$$\tau_1 \doteq \tau_2$$

In general, a **type constraint** is a predicate on types. The only kind we will consider:

"τ_1 should be the same as τ_2"

Enforcing a constraint like this is called **unifying** τ_1 and τ_2

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

Practice Problem

$$\{f : \alpha \to \alpha\} \vdash f\ (f\ 2 = 2) :\ \tau \dashv \mathscr{C}$$

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

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

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

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

Answer

$$\{f : \alpha \to \alpha\} \vdash f\ (f\ 2 = 2) : \ \tau \dashv \mathscr{C}$$

Let-Expressions

HM$^-$ (Typing Let-Expressions)

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

```
The type of a let-expression is the same as the type of its body,
relative to the constraints of typing the let-binding and the body
(wordy...)
```

Let-Polymorphism

```
let f = fun x -> x in
let y = f 2 in
f true
```

Let-Polymorphism

```
let f = fun x -> x in
let y = f 2 in
f true
```

<u>The "Problem":</u> This rule does not allow let-defined functions to be polymorphic!

Let-Polymorphism

```
let f = fun x -> x in
let y = f 2 in
f true
```

<u>The "Problem":</u> This rule does not allow let-defined functions to be polymorphic!

(This is why we call our system Hindley–Milner *Light*)

Let-Polymorphism

```
let f = fun x -> x in
let y = f 2 in
f true
```

The "Problem": This rule does not allow let-defined functions to be polymorphic!

(This is why we call our system Hindley–Milner *Light*)

There are some interesting debates in the world of PL with regards to let-polymorphism...

Let-Polymorphism

```
let f = fun x -> x in
let y = f 2 in
f true
```

The "Problem": This rule does not allow let-defined functions to be polymorphic!

(This is why we call our system Hindley–Milner *Light*)

There are some interesting debates in the world of PL with regards to let-polymorphism...

The Takeaway: We will have to treat typing of top-level let-expressions as *different* from local let-expressions

Unification

High Level

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

High Level

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

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

High Level

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

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* operations

High Level

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

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* operations

<u>The best way to think of it (in my opinion):</u> unification is solving a system of equations over *variables* and *ADT constructors*

ADT Unification Problem

ADT Unification Problem

(Informal) Given an ADT, we consider a **term** to be an element of the ADT possibly with variables (we can make this formal using *algebra*)

ADT Unification Problem

(Informal) Given an ADT, we consider a **term** to be an element of the ADT possibly with variables (we can make this formal using *algebra*)

A **unification problem** is a collection of equations of the form

ADT Unification Problem

(Informal) Given an ADT, we consider a **term** to be an element of the ADT possibly with variables (we can make this formal using *algebra*)

A **unification problem** is a collection of equations of the form

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

ADT Unification Problem

(Informal) Given an ADT, we consider a **term** to be an element of the ADT possibly with variables (we can make this formal using *algebra*)

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

Unifiers

Unifiers

Given a unification problem $\mathscr{U}$, a **solution** or **unifier** is a sequence of substitutions to *some* of the variables appearing in $\mathscr{U}$, typically written

Unifiers

Given a unification problem $\mathscr{U}$, a **solution** or **unifier** is a sequence of substitutions to *some* of the variables appearing in $\mathscr{U}$, typically written

$$\mathscr{S} = \{x \mapsto t_1, x \mapsto t_2, \ldots, x_n \mapsto t_n\}$$

Unifiers

Given a unification problem $\mathscr{U}$, a **solution** or **unifier** is a sequence of substitutions to *some* of the variables appearing in $\mathscr{U}$, typically written

$$\mathscr{S} = \{x \mapsto t_1, x \mapsto t_2, \ldots, x_n \mapsto t_n\}$$

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

Unifiers

Given a unification problem $\mathscr{U}$, a **solution** or **unifier** is a sequence of substitutions to *some* of the variables appearing in $\mathscr{U}$, typically written

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

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

Unifiers

Given a unification problem $\mathcal{U}$, a **solution** or **unifier** is a sequence of substitutions to *some* of the variables appearing in $\mathcal{U}$, typically written

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

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

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

The Simple Case: Variables

Given a system of equations over *just* variables, the unification problem is equivalent to the **connected components** problem over undirected graphs

Type Unification

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

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

Example

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

Unification may Fail

Not all unification problems have solutions:

Most General Unifiers

Most General Unifiers

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

Most General Unifiers

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

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

Most General Unifiers

The **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*

<u>Ex.</u>

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

An Algorithm (High Level)

An Algorithm (High Level)

We process equations one at a time, updating *the collection of equations*. We **FAIL** if we reach an unsatisfiable equations

An Algorithm (High Level)

We process equations one at a time, updating *the collection of equations*. We **FAIL** if we reach an unsatisfiable equations

There are three kinds of *satisfiable* equations:

An Algorithm (High Level)

We process equations one at a time, updating *the collection of equations*. We **FAIL** if we reach an unsatisfiable equations

There are three kinds of *satisfiable* equations:

- syntactical equality (e.g., int $\doteq$ int)

An Algorithm (High Level)

We process equations one at a time, updating *the collection of equations*. We **FAIL** if we reach an unsatisfiable equations

There are three kinds of *satisfiable* equations:

» syntactical equality (e.g., int $\doteq$ int)

» function type equality (e.g., $\alpha \to \beta \doteq \alpha \to \gamma$)

An Algorithm (High Level)

We process equations one at a time, updating *the collection of equations*. We **FAIL** if we reach an unsatisfiable equations

There are three kinds of *satisfiable* equations:

» syntactical equality (e.g., $\texttt{int} \doteq \texttt{int}$)

» function type equality (e.g., $\alpha \to \beta \doteq \alpha \to \gamma$)

» assignment (e.g., $\alpha \doteq \texttt{int} \to \beta$)

An Algorithm (High Level)

We process equations one at a time, updating *the collection of equations*. We **FAIL** if we reach an unsatisfiable equations

There are three kinds of *satisfiable* equations:

» syntactical equality (e.g., int $\doteq$ int)

» function type equality (e.g., $\alpha \to \beta \doteq \alpha \to \gamma$)

» assignment (e.g., $\alpha \doteq \text{int} \to \beta$)

When we see an assignment, it *becomes part of our solution*

An Algorithm (High Level)

We process equations one at a time, updating *the collection of equations*. We **FAIL** if we reach an unsatisfiable equations

There are three kinds of *satisfiable* equations:

- » syntactical equality (e.g., int $\doteq$ int)
- » function type equality (e.g., $\alpha \to \beta \doteq \alpha \to \gamma$)
- » assignment (e.g., $\alpha \doteq \text{int} \to \beta$)

When we see an assignment, it *becomes part of our solution*

And we're guaranteed to get the a most general unifier

An Algorithm (Pseudocode)

An Algorithm (Pseudocode)

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

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

```
𝒮 ← empty solution

WHILE eq ∈ 𝒰:  // 𝒰 is not empty

  MATCH eq:
```

An Algorithm (Pseudocode)

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

```
𝒮 ← empty solution

WHILE eq ∈ 𝒰:   // 𝒰 is not empty

  MATCH eq:

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

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

MATCH eq:

$t_1 \doteq t_2 \implies \mathscr{U} \leftarrow \mathscr{U} \backslash \{eq\}$ // if t_1 and t_2 are *syntactically* equal then remove eq from $\mathscr{U}$)

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

An Algorithm (Pseudocode)

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

```
𝒮 ← empty solution

WHILE eq ∈ 𝒰:   // 𝒰 is not empty

  MATCH eq:

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

    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
```

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:

$t_1 \doteq t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\}$ // if t_1 and t_2 are *syntactically* equal then remove eq from $\mathcal{U}$)

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

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:

    $t_1 \doteq t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\}$ // if t_1 and t_2 are *syntactically* equal then remove eq from $\mathcal{U}$)

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

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

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:

    $t_1 \doteq t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\}$ // if t_1 and t_2 are *syntactically* equal then remove eq from $\mathcal{U}$)

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

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:

    $t_1 \doteq t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\}$ // if t_1 and t_2 are *syntactically* equal then remove eq from $\mathcal{U}$)

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

    $\alpha \doteq t$ or $t \doteq \alpha$ where $\alpha \notin \mathrm{FV}(t) \implies$ // 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)

<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:

    $t_1 \doteq t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\}$ // if t_1 and t_2 are *syntactically* equal then remove eq from $\mathcal{U}$)

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

RETURN $\mathcal{S}$

Example

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

Another Example

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

Principle Type

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

Principle Type

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

The constraints $\mathscr{C}$ defined a *unification problem*

Principle Type

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

The constraints $\mathcal{C}$ defined a *unification problem*

Given a unifier $\mathcal{S}$ for $\mathcal{C}$ we can get the "actual" type of e:

Principle Type

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

The constraints $\mathcal{C}$ defined a *unification problem*

Given a unifier $\mathcal{S}$ for $\mathcal{C}$ we can get the "actual" type of e:

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

Principle Type

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

The constraints $\mathcal{C}$ defined a *unification problem*

Given a unifier $\mathcal{S}$ for $\mathcal{C}$ we can get the "actual" type of e:

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

This is called the **principle type** of e. Every type we *could* give e is a *specialization* $\forall \alpha_1, \ldots, \alpha_k . \mathcal{S}\tau$

Example

Determine the principle type of **`fun f -> fun x -> f (x + 1)`**

Example

Show that $f\ (f\ 2 = 2)$ *has no principle type in the context* $\{f : \alpha \to \alpha\}$

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)`

$\Gamma \leftarrow \varnothing$

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**:**

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

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 unifier $\mathscr{S}$ (throw a **TYPE ERROR** if this fails)

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 unifier $\mathscr{S}$ (throw a **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 unifier $\mathscr{S}$ (throw a **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

Summary

Unification is used to solve a collection of constraints generated by constraint-based inference

Not all unification problems have solutions. In the type unification problem, this indicates a type error

The **principle type** of an expression is the most general type we could give to an expression in our system