# Unification

## Concepts of Programming Languages

CAS CS 320

# Outline

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

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

**monotype ($\sigma$):** type with no quantification

**monomorphic type:** monotype with no type variables

**type scheme ($\tau$):** type with zero or more quantified type variables

**polymorphic type:** *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

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

# Recall: Constraints

$$\tau_1 \doteq \tau_2$$

"$\tau_1$ should be the same as $\tau_2$"

Enforcing this constraint means **unifying** $\tau_1$ and $\tau_2$

# Recall: HM$^-$ (Typing)

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

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

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

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

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

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

# Practice Problem

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

*Determine the type $\tau$ 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

$$\{f : \alpha \rightarrow \alpha\} \vdash f\ (f\ 2 = 2) :$$

# HM$^-$ (Typing Variables)

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

# $\text{HM}^{-}$ (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 $\Gamma$, then $x$ can be given the type $\tau$ *with all free variables replaced by* ***fresh variables***

# HM$^-$ (Typing Variables)

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

If $x$ is declared in $\Gamma$, then $x$ can be given the type $\tau$ *with all free variables replaced by **fresh variables***

*This is where the polymorphism magic happens*

# $\text{HM}^-$ (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 $\Gamma$, then $x$ can be given the type $\tau$ *with all free variables replaced by* ***fresh variables***

*This is where the polymorphism magic happens*

**fresh variables can be unified with anything**

# Example

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

# 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...)

# Aside: Let-Polymorphism

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

# Aside: 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!

# Aside: 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*)

# Aside: 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...

# Aside: 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

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

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

# 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

e.g., we could solve a system of equations over *variables* and *ADT constructors*

# ADT Unification Problem

# ADT Unification Problem

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

# ADT Unification Problem

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

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

# 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 element of the ADT possibly with variables

# Example

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

# Unifiers (1)

# Unifiers (1)

A **solution** or **unifier** is a sequence of substitutions to *some* of the variables appearing in the unification problem $\mathscr{U}$:

# Unifiers (1)

A **solution** or **unifier** is a sequence of substitutions to *some* of the variables appearing in the unification problem $\mathscr{U}$:

$$\mathscr{S} = \{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_i \mapsto t_i\}$$

# Unifiers (1)

A **solution** or **unifier** is a sequence of substitutions to *some* of the variables appearing in the unification problem $\mathscr{U}$:

$$\mathcal{S} = \{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_i \mapsto t_i\}$$

We write $\mathcal{S}t$ for $[t_i/x_i]\ldots[t_1/x_1]t$

# Unifiers (2)

A solution must have the property that it **satisfies** every equation

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

# Example

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

# Unification may Fail

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

`Not all unification problems have solutions...`

# Most General Unifiers

# Most General Unifiers

A **most general unifier** 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

A **most general unifier** 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

A **most general unifier** 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)

Process each equation, updating *the collection of equations*, **FAIL** if we reach an unsatisfiable equation

# An Algorithm (High Level)

Process each equation, updating *the collection of equations*, **FAIL** if we reach an unsatisfiable equation

There are three kinds of *satisfiable* equations:

# An Algorithm (High Level)

Process each equation, updating *the collection of equations*, **FAIL** if we reach an unsatisfiable equation

There are three kinds of *satisfiable* equations:

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

# An Algorithm (High Level)

Process each equation, updating *the collection of equations*, **FAIL** if we reach an unsatisfiable equation

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)

Process each equation, updating *the collection of equations*, **FAIL** if we reach an unsatisfiable equation

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., $\alpha \doteq \text{int} \to \beta$)

# An Algorithm (High Level)

Process each equation, updating *the collection of equations*, **FAIL** if we reach an unsatisfiable equation

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)

Process each equation, updating *the collection of equations*, **FAIL** if we reach an unsatisfiable equation

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., $\alpha \doteq \mathtt{int} \to \beta$)

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

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

# 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)

<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 𝒰
output: most general unifier to 𝒰

𝒮 ← empty solution

WHILE eq ∈ 𝒰:  // 𝒰 is not empty

  MATCH eq:

    t₁ ≐ t₂ when t₁ = t₂ ⟹ 𝒰 ← 𝒰\{eq}  // t₁ and t₂ are syntactically equal, 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 when t₁ = t₂  ⟹  𝒰 ← 𝒰\{eq}   // t₁ and t₂ are syntactically equal, 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)

```
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 when t₁ = t₂ ⟹ 𝒰 ← 𝒰\{eq}  // t₁ and t₂ are syntactically equal, 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
```

# 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 when t₁ = t₂ ⟹ 𝒰 ← 𝒰\{eq}   // t₁ and t₂ are syntactically equal, 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 $\mathscr{U}$
<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** $eq$:

    $t_1 \doteq t_2$ when $t_1 = t_2 \implies \mathscr{U} \leftarrow \mathscr{U} \backslash \{eq\}$ // $t_1$ and $t_2$ are *syntactically* equal, remove $eq$

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

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

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

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

# 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** $eq$:

    $t_1 \doteq t_2$ when $t_1 = t_2 \Longrightarrow \mathcal{U} \leftarrow \mathcal{U} \backslash \{eq\}$ // $t_1$ and $t_2$ are *syntactically* equal, 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 $\alpha$ 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 $\mathscr{U}$
<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** $eq$:

$t_1 \doteq t_2$ when $t_1 = t_2 \implies \mathscr{U} \leftarrow \mathscr{U} \backslash \{eq\}$ // $t_1$ and $t_2$ are *syntactically* equal, remove $eq$

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

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

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

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

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

**OTHERWISE** $\implies$ **FAIL**

# 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 when t₁ = t₂ ⟹ 𝒰 ← 𝒰\{eq}   // t₁ and t₂ are syntactically equal, 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

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

# Example

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

# 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