

Unification

Concepts of Programming Languages Lecture 22

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

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

Recall: Type Schemes

$$\sigma ::= \text{int} \mid \text{bool} \mid \alpha \mid \sigma \rightarrow \sigma$$

$$\tau ::= \sigma \mid \forall \alpha. \tau$$

monotype

$$\forall \alpha_1 \dots \forall \alpha_k. (\sigma_1 \rightarrow \text{bool}) \rightarrow \alpha_k$$

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

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

Our typing rules will need to keep track of a set of **constraints**, which tell us 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)

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \text{int} \dashv \emptyset} \text{ (int)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \quad \Gamma \vdash e_3 : \tau_3 \dashv \mathcal{C}_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_3 \dashv \tau_1 \doteq \text{bool}, \tau_2 \doteq \tau_3, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3} \text{ (if)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 = e_2 : \text{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \text{ (eq)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 + e_2 : \text{int} \dashv \tau_1 \doteq \text{int}, \tau_2 \doteq \text{int}, \mathcal{C}_1, \mathcal{C}_2} \text{ (add)}$$

$$\frac{\boxed{\alpha \text{ is fresh}} \quad \Gamma, x : \alpha \vdash e : \tau \dashv \mathcal{C}}{\Gamma \vdash \lambda x. e : \alpha \rightarrow \tau \dashv \mathcal{C}} \text{ (fun)}$$

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

Recall: HM⁻ (Typing Variables)

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

monotype (pointing to τ)

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

not polymorphic

$$\{f : \boxed{\alpha \rightarrow \alpha}\} \vdash f (f \ 2 = 2) : \tau \dashv \mathcal{C}$$

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

$$\frac{n \text{ is an integer}}{\Gamma \vdash n : \text{int} \dashv \emptyset} \quad (\text{int})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash e_1 = e_2 : \text{bool} \dashv \tau_1 \doteq \tau_2, \mathcal{C}_1, \mathcal{C}_2} \quad (\text{eq})$$

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

Answer

$$\{f: \alpha \rightarrow \alpha\} \vdash f(f\ 2 = 2) : \tau \dashv \mathcal{C}$$

$$\{f: \alpha \rightarrow \alpha\} \vdash f(f\ 2 = 2) : \boxed{\gamma} \dashv \alpha \rightarrow \alpha \doteq \text{bool} \rightarrow \boxed{\gamma}, \beta \doteq \text{int}, \alpha \rightarrow \alpha \doteq \text{int} \rightarrow \beta$$

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

$$\left\{ \begin{array}{l} \{f: \alpha \rightarrow \alpha\} \vdash f\ 2 = 2 : \text{bool} \dashv \boxed{\beta} \doteq \boxed{\text{int}}, \alpha \rightarrow \alpha \doteq \text{int} \rightarrow \beta \end{array} \right.$$

$$\left\{ \begin{array}{l} \{f: \alpha \rightarrow \alpha\} \vdash f\ 2 : \boxed{\beta} \dashv \alpha \rightarrow \alpha \doteq \text{int} \rightarrow \beta \end{array} \right.$$

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

$$\left\{ \begin{array}{l} \{f: \alpha \rightarrow \alpha\} \vdash 2 : \text{int} \dashv \emptyset \end{array} \right.$$

$$\left\{ \begin{array}{l} \{f: \alpha \rightarrow \alpha\} \vdash 2 : \boxed{\text{int}} \dashv \emptyset \end{array} \right.$$

$$C = \underline{\alpha} \rightarrow \alpha \doteq \underline{\text{bool}} \rightarrow \gamma$$

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

$$\underline{\alpha} \rightarrow \alpha \doteq \underline{\text{int}} \rightarrow \beta$$

Let-Expressions

HM⁻ (Typing Let-Expressions)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \dashv \mathcal{C}_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \dashv \mathcal{C}_1, \mathcal{C}_2} \quad (\text{let})$$

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
```

The "Problem": 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
```

The "Problem": 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

$$a \doteq d \rightarrow e$$

$$c \doteq \text{int} \rightarrow d$$

$$\text{int} \rightarrow \text{int} \rightarrow \text{int} \doteq b \rightarrow c$$

High Level

$$a \doteq d \rightarrow e$$

$$c \doteq \text{int} \rightarrow d$$

$$\text{int} \rightarrow \text{int} \rightarrow \text{int} \doteq b \rightarrow c$$

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

High Level

$$a \doteq d \rightarrow e$$

$$c \doteq \text{int} \rightarrow d$$

$$\text{int} \rightarrow \text{int} \rightarrow \text{int} \doteq b \rightarrow c$$

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

$$a \doteq d \rightarrow e$$

$$c \doteq \text{int} \rightarrow d$$

$$\text{int} \rightarrow \text{int} \rightarrow \text{int} \doteq b \rightarrow c$$

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

The best way to think of it (in my opinion): 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*)

Nil

$\text{Cons} (1, \text{Cons} (2, \text{Nil}))$

$\text{Cons} (1, x)$

y

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

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

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

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

where s_1, \dots, s_k and t_1, \dots, t_k are terms

Unifiers


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

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

$$\mathcal{S} = \{x_{\underline{1}} \mapsto t_1, x_{\underline{2}} \mapsto t_2, \dots, x_n \mapsto t_n\}$$

ordered

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

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

We write $\mathcal{S}t$ for $\underbrace{[t_n/x_n] \dots [t_1/x_1]t}_{\text{reverse order}}$

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

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

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

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

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

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

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

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

not \doteq , but syntactic equal

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

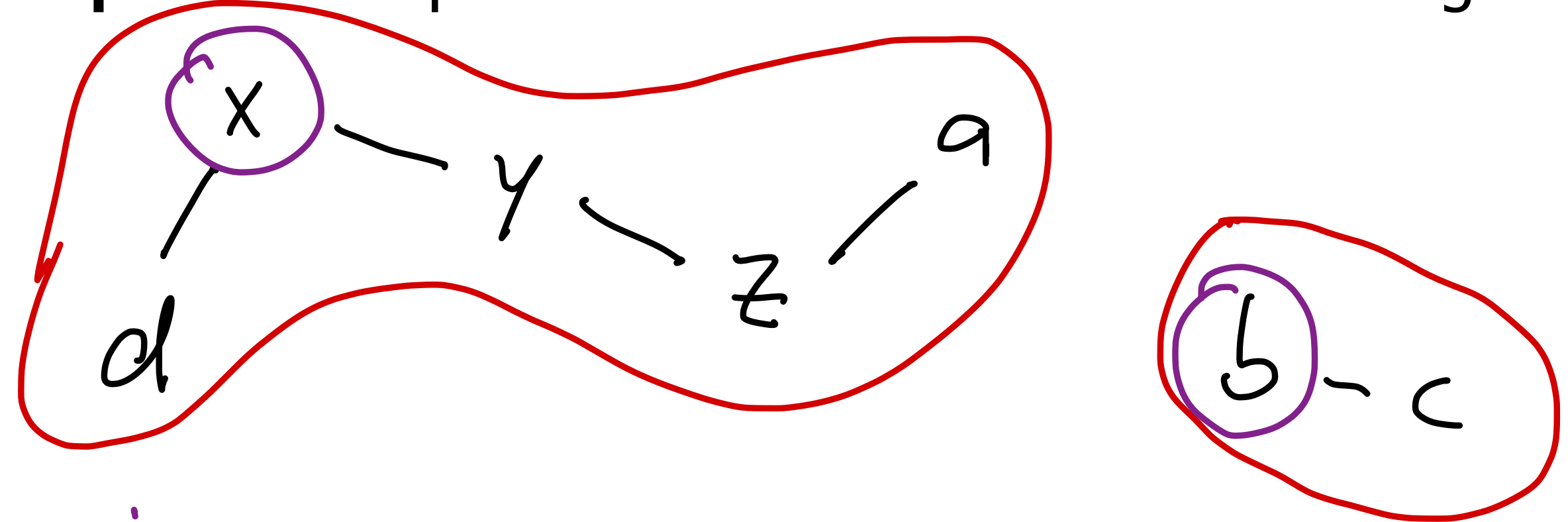
$$x \doteq y$$

$$y \doteq z$$

$$a \doteq z$$

$$b \doteq c$$

$$d \doteq x$$



$$S = \{y \mapsto x, z \mapsto x, a \mapsto x, d \mapsto x, c \mapsto b\}$$

$$S_x = x = x = S_y \quad \checkmark$$

$$S_y = x = x = S_z$$

⋮

Type Unification

```
type ty =
```

```
| TInt
```

```
| TBool
```

```
| TFun of ty * ty
```

```
| TVar of string
```

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

$$\beta \rightarrow \beta \doteq \text{bool} \rightarrow \text{bool}$$

$$\gamma \doteq \alpha$$

$$S = \left\{ \begin{array}{l} \beta \mapsto \text{bool} \\ \alpha \mapsto \text{int} \rightarrow \text{bool} \\ \gamma \mapsto \text{int} \rightarrow \text{bool} \end{array} \right\}$$

$$S\alpha = \text{int} \rightarrow \text{bool} = \text{int} \rightarrow \text{bool} = S(\text{int} \rightarrow \beta)$$

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

Example

$$a \doteq d \rightarrow e$$

$$c \doteq \text{int} \rightarrow d$$

$$\text{int} \rightarrow \text{int} \rightarrow \text{int} \doteq b \rightarrow c$$

$$S = \left\{ \begin{array}{l} b \mapsto \text{int} \\ c \mapsto \text{int} \rightarrow \text{int} \\ d \mapsto \text{int} \\ a \mapsto \text{int} \rightarrow e \end{array} \right\}$$

$e \mapsto \text{bool}$

$$S_a = \text{int} \rightarrow e = \text{int} \rightarrow e = S(\text{int} \rightarrow e) \checkmark$$

$$S_c = \text{int} \rightarrow \text{int} = \text{int} \rightarrow \text{int} = S(\text{int} \rightarrow d) \checkmark$$

$$\text{int} \rightarrow \text{int} \rightarrow \text{int} = \text{int} \rightarrow \text{int} \rightarrow \text{int} \checkmark$$

Unification may Fail

Not all unification problems have solutions:

$$\text{int} \doteq \text{bool} \quad \times$$

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

$$\beta \rightarrow \alpha = \text{int} \rightarrow \text{bool} \quad \times$$

$$\alpha \doteq \alpha \rightarrow \text{int} \quad \times$$

Most General Unifiers

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 $\mathcal{S}' = \mathcal{S}\mathcal{S}''$

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*

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

Ex.

$$a \doteq d \rightarrow e$$

$$c \doteq \text{int} \rightarrow d$$

$$\text{int} \rightarrow \text{int} \rightarrow \text{int} \doteq b \rightarrow c$$

$$\text{mgu } \mathcal{S} = \begin{cases} b \mapsto \text{int} \\ c \mapsto \text{int} \rightarrow \text{int} \\ d \mapsto \text{int} \\ a \mapsto \text{int} \rightarrow e \end{cases}$$

$$\mathcal{S}' = \mathcal{S} \cup \{e \mapsto \text{bool}\}$$

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., $\text{int} \doteq \text{int}$)

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., $\text{int} \doteq \text{int}$)

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

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

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

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., $\text{int} \doteq \text{int}$)

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

» assignment (e.g., $\alpha \doteq \overbrace{\text{int} \rightarrow \beta}^{\text{type}}$)
 $\underbrace{\alpha}_{\text{type var}}$

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 \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 :

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

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

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// 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 :

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

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// 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 :

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

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// 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 :

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

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// 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 :

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

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ // 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} *no more references to α*

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 :

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

$s_1 \rightarrow t_1 \doteq s_2 \rightarrow t_2 \implies \mathcal{U} \leftarrow \mathcal{U} \setminus \{eq\} \cup \{s_1 \doteq s_2, t_1 \doteq t_2\}$ *// 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 :

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

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

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

$\alpha \notin \text{FV}(\text{int} \rightarrow \alpha)$

Example

$$\cancel{a \dot{=} d \rightarrow e} \quad \alpha \dot{=} t$$

$$\cancel{c \dot{=} \text{int} \rightarrow d} \quad \alpha \dot{=} t$$

$$\cancel{\text{int} \rightarrow \text{int} \rightarrow \text{int} \dot{=} b \rightarrow c \text{ int} \rightarrow d}$$

$$\cancel{\text{int} \dot{=} b} \quad t = \alpha$$

$$\cancel{\text{int} \rightarrow \text{int} \dot{=} \text{int} \rightarrow d}$$

$$\cancel{\text{int} \dot{=} d} \quad t \dot{=} \alpha$$

$$\cancel{\text{int} \dot{=} \text{int}}$$

$$S = \begin{cases} a \mapsto d \rightarrow e \\ c \mapsto \text{int} \rightarrow d \\ b \mapsto \text{int} \\ d \mapsto \text{int} \end{cases}$$

$$s_1 \rightarrow t_1 \dot{=} s_2 \rightarrow t_2$$

$$\rightarrow \dot{=} \rightarrow$$

Another Example

$$\beta \doteq \eta$$

$$\alpha \rightarrow \beta \doteq \alpha \rightarrow \gamma$$

$$\alpha \rightarrow \beta \doteq \gamma \rightarrow \eta$$

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

Principle Type

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

Principle Type

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

The constraints \mathcal{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 :

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

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 :

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

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

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 $\{f: \alpha \rightarrow \alpha\}$

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)

$\Gamma \leftarrow \emptyset$

Putting everything together

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

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

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

Putting everything together

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 unifier \mathcal{S} (throw a **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

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 unifier \mathcal{S} (throw a **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

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