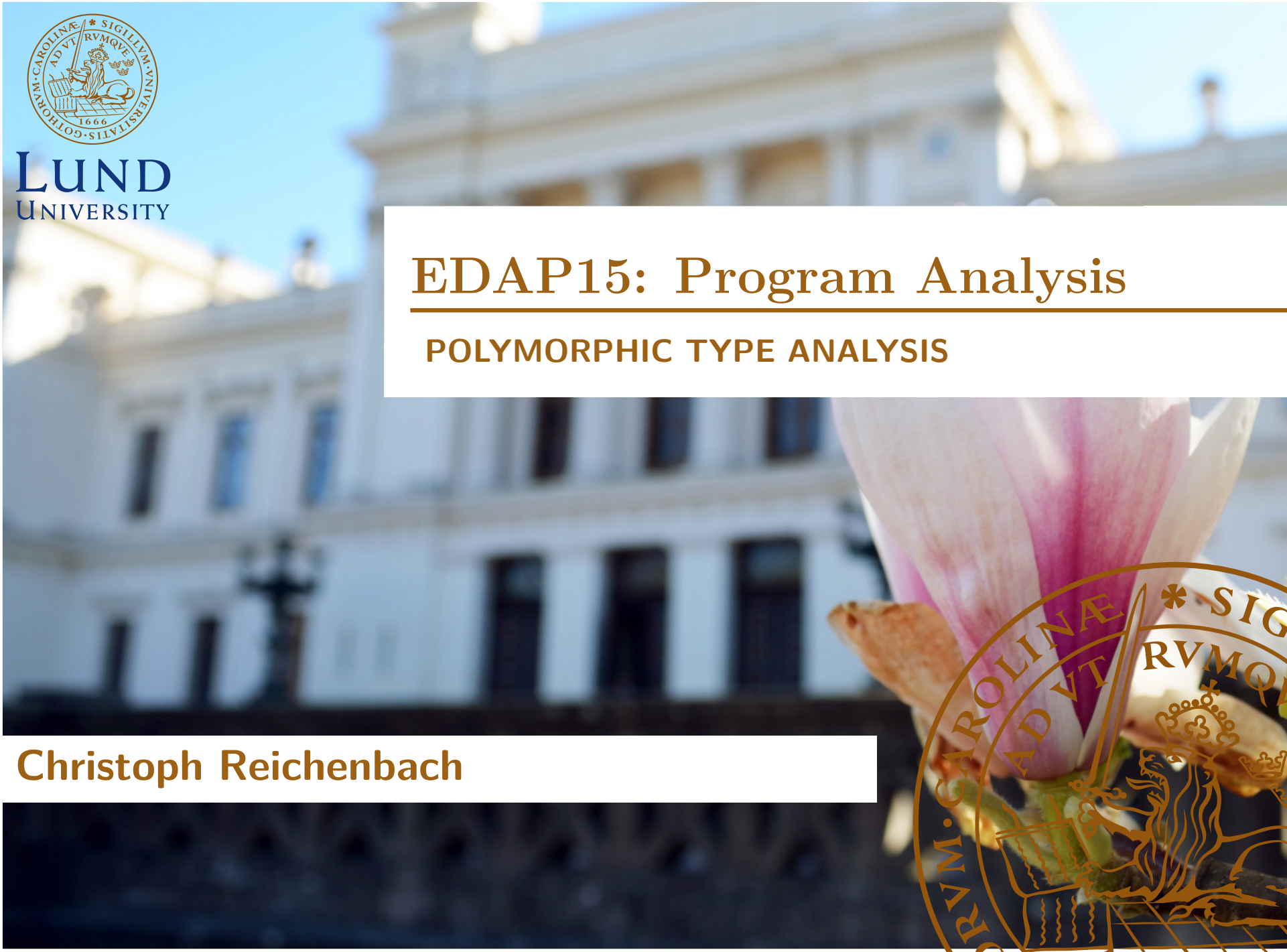


LUND UNIVERSITY

EDAP15: Program Analysis

POLYMORPHIC TYPE ANALYSIS

Christoph Reichenbach

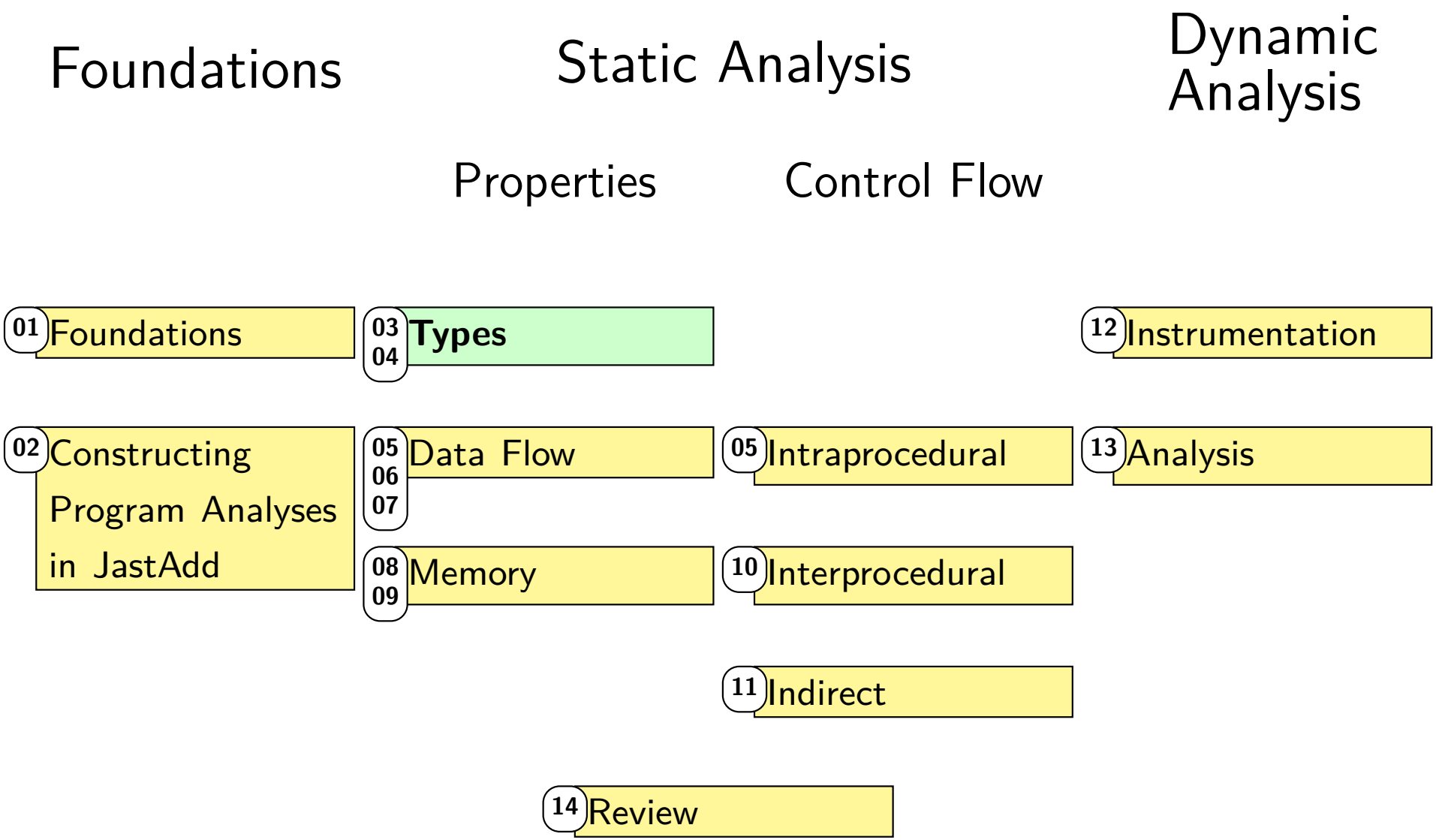
Announcements

- Current lab group distribution:

Time	Groups
10:00	7
13:00	4
15:00	1

- Lab 1 is out

Lecture Overview



Parametric Types and Principal Types

$$
\begin{array}{rcl}
\mathit{Ty} & ::= & \textsc{Int} \\
& | & \textsc{Bool} \\
& | & \textsc{List}\ [\langle \mathit{Ty} \rangle]
\end{array}
$$

- Challenge:
  - `nil` : $\textsc{List}[\tau]$ allows infinitely many types
- To make practical: find **Principal Type**
  - *Single* type that summarises all other types
- Here: use **Parametric Types** with **Type Variables**:
  - $\textsc{List}[\alpha]$ summarises $\textsc{List}[\textsc{Int}]$, $\textsc{List}[\textsc{Bool}]$, $\textsc{List}[\textsc{List}[\ldots]]$
- Difference τ, α when computing type:
  - τ: must solve when we see AST node
  - α: is a valid type already
    - We can "refine" it (substitute something else) *later*

Typing Rules: Parametric Types

— First attempt —

$$\frac{}{\text{true} : \textsc{Bool}}\ (t\text{-}true) \qquad \frac{}{\text{false} : \textsc{Bool}}\ (t\text{-}false) \qquad \frac{v \in Nat}{v : \textsc{Int}}\ (t\text{-}nat)$$

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}}\ (t\text{-}plus) \qquad \frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 >= e_2 : \textsc{Bool}}\ (t\text{-}ge) \qquad \frac{\Delta(\underline{x}) = \tau}{\underline{x} : \tau}\ (t\text{-}var)$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\ (t\text{-}if) \qquad \frac{e_1 : \tau_1 \quad \Delta(\underline{x}) = \tau_1 \quad e_2 : \tau_2}{\text{let } \underline{x} = e_1 \text{ in } e_2 : \tau_2}\ (t\text{-}let)$$

$$\frac{}{\text{nil} : \textsc{List}[\alpha]}\ (t\text{-}nil) \qquad \frac{e_1 : \tau \quad e_2 : \textsc{List}[\tau]}{\text{cons}(e_1, e_2) : \textsc{List}[\tau]}\ (t\text{-}cons)$$

Originally $\Delta(\underline{x}) = \textsc{List}[\alpha]$

Must merge $\textsc{List}[\alpha] = \textsc{List}[\textsc{Int}]$

Analogous to variable types

$\Delta(\underline{x}) = \cancel{\textsc{List}[\alpha]}\ \textsc{List}[\textsc{Int}]$

$\boxed{\Delta(\alpha) = \textsc{Int}}$

$$\frac{\dfrac{}{\text{nil} : \textsc{List}[\alpha]}\ (t\text{-}nil) \qquad \Delta(\underline{x}) = \textsc{List}[\alpha] \qquad \dfrac{\dfrac{1 \in Nat}{1 : \textsc{Int}}\ (t\text{-}nat) \qquad \dfrac{\Delta(\underline{x}) = \textsc{List}[\textsc{Int}]}{\underline{x} : \textsc{List}[\textsc{Int}]}}{\text{cons}(1, \underline{x}) : \textsc{List}[\textsc{Int}]}}{\text{let } \underline{x} = \text{nil} \text{ in } \text{cons}(1, \underline{x}) : \textsc{List}[\textsc{Int}]}\ (t\text{-}let)$$

Typing Rules: Parametric Types

— First attempt —

$$\frac{}{\text{true} : \textsc{Bool}}\ (t\text{-}true) \qquad \frac{}{\text{false} : \textsc{Bool}}\ (t\text{-}false) \qquad \frac{v \in Nat}{v : \textsc{Int}}\ (t\text{-}nat)$$

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}}\ (t\text{-}plus) \qquad \frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 >= e_2 : \textsc{Bool}}\ (t\text{-}ge) \qquad \frac{\Delta(x) = \tau}{x : \tau}\ (t\text{-}var)$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\ (t\text{-}if) \qquad \frac{e_1 : \tau_1 \quad \Delta(x) = \tau_1 \quad e_2 : \tau_2}{\text{let } x = e_1 \text{ in } e_2 : \tau_2}\ (t\text{-}let)$$

$$\frac{}{\text{nil} : \textsc{List}[\alpha]}\ (t\text{-}nil) \qquad \frac{e_1 : \tau \quad e_2 : \textsc{List}[\tau]}{\text{cons}(e_1, e_2) : \textsc{List}[\tau]}\ (t\text{-}cons)$$

Circular type — *t-cons* requires:

$$\begin{array}{lcl} \tau & = & \textsc{List}[\alpha] \\ \textsc{List}[\tau] & = & \textsc{List}[\alpha] \end{array}$$

$\Delta(\alpha) = \textsc{List}[\alpha]$

$$\frac{\dfrac{}{\text{nil} : \textsc{List}[\alpha]}\ (t\text{-}nil) \qquad \dfrac{}{\text{nil} : \textsc{List}[\alpha]}\ (t\text{-}nil)}{\text{cons}(\text{nil}, \text{nil}) : \ ?}\ (t\text{-}cons)$$

Type Variable Freshness

- Our typing rule for nil doesn't work as intended:
  All nil use the same α in their type
  $\Longrightarrow$ all lists must have the same type

$$\frac{\alpha \quad \textit{fresh}}{\text{nil} : \textsc{List}[\alpha\alpha]}\ (\textit{t-nil}) \qquad \frac{e_1 : \tau \quad e_2 : \textsc{List}[\tau]}{\text{cons}(e_1,\ e_2) : \textsc{List}[\tau]}\ (\textit{t-cons})$$

- Fix: We create a *fresh* type variable for every nil
  (specific to AST node, different from all others)

$\boxed{\Delta(\alpha_1) = (\textit{not set})}$

$\boxed{\Delta(\alpha_2) = (\textit{not set})\textsc{List}[\alpha_1]}$

$$\frac{\dfrac{\alpha_1 \quad \textit{fresh}}{\text{nil} : \tau\textsc{List}[\alpha_1]}\ (\textit{t-nil}) \qquad \dfrac{\alpha_2 \quad \textit{fresh}}{\text{nil} : \textsc{List}[\tau\textsc{List}[\alpha_1]]\textsc{List}[\alpha_2]}\ (\textit{t-nil})}{\text{cons}(\text{nil},\ \text{nil}) : \textsc{List}[\textsc{List}[\alpha_1]]}\ (\textit{t-cons})$$

Typing Rules: Parametric Types

Fixed version

$$\frac{}{\text{true} : \textsc{Bool}}\ (t\text{-}true) \qquad \frac{}{\text{false} : \textsc{Bool}}\ (t\text{-}false) \qquad \frac{v \in Nat}{v : \textsc{Int}}\ (t\text{-}nat)$$

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \text{ plus } e_2 : \textsc{Int}}\ (t\text{-}plus) \qquad \frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 >= e_2 : \textsc{Bool}}\ (t\text{-}ge) \qquad \frac{\Delta(\underline{x}) = \tau}{\underline{x} : \tau}\ (t\text{-}var)$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}\ (t\text{-}if) \qquad \frac{e_1 : \tau_1 \quad \Delta(\underline{x}) = \tau_1 \quad e_2 : \tau_2}{\text{let } \underline{x} = e_1 \text{ in } e_2 : \tau_2}\ (t\text{-}let)$$

$$\frac{\alpha \quad fresh}{\text{nil} : \textsc{List}[\alpha]}\ (t\text{-}nil) \qquad \frac{e_1 : \tau \quad e_2 : \textsc{List}[\tau]}{\text{cons}(e_1, e_2) : \textsc{List}[\tau]}\ (t\text{-}cons)$$

$$\boxed{\Delta(\alpha_1) = (not\ set)}$$

$$\boxed{\Delta(\alpha_2) = \textsc{List}[\alpha_1]}$$

$$\frac{\dfrac{\alpha_1 \quad fresh}{\text{nil} : \textsc{List}[\alpha_1]}\ (t\text{-}nil) \qquad \dfrac{\alpha_2 \quad fresh}{\text{nil} : \textsc{List}[\textsc{List}[\alpha_1]]}\ (t\text{-}nil)}{\text{cons}(\text{nil}, \text{nil}) : \textsc{List}[\textsc{List}[\alpha_1]]}\ (t\text{-}cons)$$

Summary

- Some expressions may have multiple or even unbounded number of types

$\Rightarrow$ *Polymorphic expression*

- With **parametric types**: use **type variables** to present such types

$\Rightarrow$ *Polymorphic type*

- Also want this to be a **principal type**:
  - Summarise *all* possible types
- Polymorphic types may *specialise* / *instantiate*:
  - Substitute type variable by more concrete type
  - **Same polymorphic values can instantiate more than once**
    - Different substitutions
    - Standard solution: use **fresh type variables** for every distinct use
- Still need to solve:
  How *do* we merge type variables in equations?

$$\textsc{List}[\alpha_1] = \textsc{List}[\textsc{List}[\alpha_2]]$$

Parametric Types in Practice

- Widely used today, e.g. *Generics* in Java:

Java	**Scala**
`List<E>`	`List[A]`
`Set<E>`	`Set[A]`
`Map<K, V>`	`Map[K, V]`

- Also used as the type of *functions*:

Java	**Scala**	**Common**
`Function<T, R>`	`A => B`	$\alpha \to \beta$

- Scala and others also support parametric *tuple types*:

Scala	**Ocaml/SML**	**Common**
`(A, B, C)`	`'a * 'b * 'c`	$\alpha \times \beta \times \gamma$

- We often combine tuple and function types when inferring types of functions:

$$\texttt{countOccurrencesInList} : \textsc{List}[\alpha] \times \alpha \to \textsc{Int}$$

Typing Challenge 1

- Let's try to do (static) type inference for Python
  (No typing rules, focus on intuition)
- What should the type of `f` be:

Python

```
def f(x:int) :
    return [x]
```

- `f`: ... INT → ... LIST[INT]
- `x` has no initialiser, but `x:int` makes type clear

Typing Challenge 2

- Let's try to do (static) type inference for Python
  (No typing rules, focus on intuition)
- What should the type of g be:

Python

```
def g(y_) :
    return [y_]
```

- g: $\ldots\alpha \rightarrow \ldots \textsc{List}[\alpha]$
- No information that helps us find type of y_!
- Using type variable captures idea:
  input type determines output type

Solving Typing Challenge 2

Python

```
def g(y) :
    return [y]
```

- Can we apply *t-var* for g's y?

$$\frac{\Delta(x) = \tau}{x : \tau}\ (t\text{-}var)$$

- **No:** Type of y never defined in this code

$\Rightarrow$ no τ exists for $\Delta(x) = \tau$

$\Rightarrow$ Cannot satisfy premise of rule *t-var*

- If we don't know the concrete type, anything is possible
- So let's assign a fresh type variable as type of each variable:

$$\frac{\Delta(x) = \alpha \quad \alpha\ \mathit{fresh}}{x : \alpha}\ (t\text{-}var')$$

Typing Variables with Type Variables

$$\frac{\Delta(\underline{x}) = \alpha \quad \alpha \textit{ fresh}}{\underline{x} : \alpha}\ (\textit{t-var}')$$

Teal-0

```
var x;
x := 1;


x := "foo";
```

$\Delta(\underline{x}) = \alpha$
$1 : \textsc{Int}$
$\alpha = \textsc{Int}$

$\Delta(\underline{x}) = \beta$
$\texttt{"foo"} : \textsc{String}$
$\beta = \textsc{String}$

- Each assignment involves variable $\underline{x}$
- Create one type variable per use of $\underline{x}$
- Observe that $\alpha = \textsc{Int}$
- Observe that $\beta = \textsc{String}$
- Note that $\alpha = \Delta(\underline{x}) = \beta$: Found type error
- Simplification: attach type variable to *declaration* of $\underline{x}$, so we only need α (no β)

Simplified use of Type Variables

$$\frac{\Delta(\underline{x}) = \alpha \quad \alpha \textit{ fresh}}{\underline{x} : \alpha}\ (\textit{t-var}')$$

Teal-0

```
var x;
x := 1;


x := "foo";
```

$\underline{x} : \alpha$
$1 : \textsc{Int}$
$\alpha = \textsc{Int}$

$\underline{x} : \alpha$
$\texttt{"foo"} : \textsc{String}$
$\alpha = \textsc{String}$

Simplified Notation

- Only one type variable per variable declaration
- Write $\underline{x} : \alpha$ instead of $\Delta(\underline{x}) = \alpha$
- Observe: two kinds of "constraints" from code:
  - Typings: $\underline{v} : \tau$
  - Equalities: $\tau_1 = \tau_2$

Type Inference with Variables: Example

Python

```
def gen(a:map, b:set):
1   m = {}
2   for v in b:
3       if v in a.keys():
4           x = a[v]
5           m[x] = x
6   return m
```

Extract *typings*:

$\mathsf{y} : \tau$

Extract *equality constraints*:

$\tau_1 = \tau_2$

$$
\begin{array}{rr}
 & \mathsf{a} : \mathtt{map}[\beta_1, \beta_2] \\
 & \mathsf{b} : \mathtt{set}[\gamma] \\
 & \mathsf{gen} : \mathtt{map}[\beta_1, \beta_2] \times \mathtt{set}[\gamma] \to \xi \\
1 & \mathsf{m} : \mathtt{map}[\alpha_1, \alpha_2] \\
2 & \mathsf{v} : \gamma \\
3 & \mathsf{v} : \beta_1 \\
 & \gamma = \beta_1 \\
4 & \mathsf{x} : \delta \\
 & \mathsf{a} : \mathtt{map}[\gamma, \delta] \\
 & \mathtt{map}[\beta_1, \beta_2] = \mathtt{map}[\gamma, \delta] \\
5 & \mathsf{m} : \mathtt{map}[\delta, \delta] \\
 & \mathtt{map}[\alpha_1, \alpha_2] = \mathtt{map}[\delta, \delta] \\
6 & \mathsf{m} : \xi \\
 & \xi = \mathtt{map}[\alpha_1, \alpha_2]
\end{array}
$$

How do we solve this automatically?

Summary

- Challenges in type analysis:
  Identifiers may not make their types conveniently accessible:
  1. Identifier type is "under-constrained", and/or
  2. Identifier type depends on itself (e.g., identifier is name of a recursive function)
- Side-step by adding indirection: Type variables $\alpha, \beta, \ldots$
- With this approach, analysing code produces two kinds of *constraints*:
  1. Typings: $\underline{\mathsf{x}} : \tau$
  2. Equality constraints: $\tau_1 = \tau_2$
- Completing type analysis requires solving these constraints

Type Inference: Constraints

Typings:

$$
\begin{array}{rcl}
\mathsf{a} & : & \texttt{map}[\beta_1, \beta_2] \\
\mathsf{b} & : & \texttt{set}[\gamma] \\
\mathsf{gen} & : & \texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\gamma] \rightarrow \xi \\
\mathsf{m} & : & \texttt{map}[\alpha_1, \alpha_2] \\
\mathsf{v} & : & \gamma \\
\mathsf{v} & : & \beta_1 \\
\mathsf{x} & : & \delta \\
\mathsf{a} & : & \texttt{map}[\gamma, \delta] \\
\mathsf{m} & : & \texttt{map}[\delta, \delta] \\
\mathsf{m} & : & \xi
\end{array}
$$

Type Equality Constraints:

$$
\begin{array}{rcl}
\gamma & = & \beta_1 \\
\texttt{map}[\beta_1, \beta_2] & = & \texttt{map}[\gamma, \delta] \\
\texttt{map}[\alpha_1, \alpha_2] & = & \texttt{map}[\delta, \delta] \\
\xi & = & \texttt{map}[\alpha_1, \alpha_2]
\end{array}
$$

Unification

$$
\begin{aligned}
\gamma &= \beta_1 \\
\texttt{map}[\beta_1, \beta_2] &= \texttt{map}[\gamma, \delta] \\
\texttt{map}[\alpha_1, \alpha_2] &= \texttt{map}[\delta, \delta] \\
\xi &= \texttt{map}[\alpha_1, \alpha_2]
\end{aligned}
$$

- *Unification* describes the problem of solving such equations
- Some unification problems are undecidable
  - *Subtyping* in particular usually leads to undecidability
- Our problem has an efficient (near-linear) solution:
  - Given a *worklist* of equality constraints:
  - Remove and process one constraint at a time
  - If constraint has form $\alpha = \tau$: replace $\alpha \mapsto \tau$
  - Otherwise, break equation into smaller equalities, add to worklist
  - . . . plus some minor tweaks

First, let us simplify our representation

Type Constructors

- Recall Parametric Types:
  - `Set`$[\alpha]$
  - `Map`$[\alpha, \beta]$
- Type constructors: things like `Set`, `Map`
  - Take type parameters α, β
  - Build new type
- Other type constructors:
  - $\cdots \times \cdots \times \cdots$: constructs product types
  - $\rightarrow$: constructs function types
- General notation: $C_i^k(\tau_1, \ldots, \tau_k)$
  - E.g.: $\texttt{int} \rightarrow \texttt{string} = C^2_{\rightarrow}(C^0_{\texttt{int}}, C^0_{\texttt{string}})$
  - E.g.: $Set[Set[\texttt{int}]] = C^1_{\texttt{Set}}(C^1_{\texttt{Set}}(C^0_{\texttt{int}}))$
- k: arity of type constructor
- i: globally unique identifier for constructor

Type Unification

- Each equation has one of these forms:
  1. $\alpha = \alpha$ (trivial)
  2. $\alpha = \beta$
     - Solution: Replace $\beta \mapsto \alpha$ everywhere
  3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
     - $\boxed{\textit{Type Error}}$ if $i \neq j$ or $k \neq l$
     - Otherwise: Replace by equations:

$$\begin{array}{lcl} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

  4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
     - Solution: Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$ everywhere
     - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots) \Rightarrow \boxed{\textit{Type Error}}$ (*"Occurs Check"*)

(Martelli and Montanari, 1982, based on Robinson, 1965)

Example (Continued)

gen : $\texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\gamma] \to \xi$

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{array}{rcl} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ *Type Error*

$$\begin{array}{rcl} \gamma & = & \beta_1 \\ \texttt{map}[\beta_1, \beta_2] & = & \texttt{map}[\gamma, \delta] \\ \texttt{map}[\alpha_1, \alpha_2] & = & \texttt{map}[\delta, \delta] \\ \xi & = & \texttt{map}[\alpha_1, \alpha_2] \end{array}$$

Example (Continued)

$$\text{gen} : \texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\beta_1] \to \xi$$

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{array}{rcl} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ *Type Error*

2. ~~$\gamma = \beta_1$~~

$$\begin{array}{rcl} \texttt{map}[\beta_1, \beta_2] & = & \texttt{map}[\beta_1, \delta] \\ \texttt{map}[\alpha_1, \alpha_2] & = & \texttt{map}[\delta, \delta] \\ \xi & = & \texttt{map}[\alpha_1, \alpha_2] \end{array}$$

Example (Continued)

gen : $\texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\beta_1] \to \xi$

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{array}{rcl} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ *Type Error*

2 ~~$\gamma = \beta_1$~~

3 ~~$\texttt{map}[\beta_1, \beta_2] = \texttt{map}[\beta_1, \delta]$~~

$$\begin{array}{rcl} \texttt{map}[\alpha_1, \alpha_2] & = & \texttt{map}[\delta, \delta] \\ \xi & = & \texttt{map}[\alpha_1, \alpha_2] \\ \beta_1 & = & \beta_1 \\ \beta_2 & = & \delta \end{array}$$

Example (Continued)

$$\text{gen} : \texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\beta_1] \to \xi$$

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{array}{rcl} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ *Type Error*

~~2~~ ~~$\gamma = \beta_1$~~

~~3~~ ~~$\texttt{map}[\beta_1, \beta_2] = \texttt{map}[\beta_1, \delta]$~~

~~3~~ ~~$\texttt{map}[\alpha_1, \alpha_2] = \texttt{map}[\delta, \delta]$~~

$$\begin{array}{rcl} \xi & = & \texttt{map}[\alpha_1, \alpha_2] \\ \beta_1 & = & \beta_1 \\ \beta_2 & = & \delta \\ \alpha_1 & = & \delta \\ \alpha_2 & = & \delta \end{array}$$

Example (Continued)

gen : $\texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\beta_1] \rightarrow \texttt{map}[\alpha_1, \alpha_2]$

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{array}{lcl} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ *Type Error*

2 ~~$\gamma = \beta_1$~~

3 ~~$\texttt{map}[\beta_1, \beta_2] = \texttt{map}[\beta_1, \delta]$~~

3 ~~$\texttt{map}[\alpha_1, \alpha_2] = \texttt{map}[\delta, \delta]$~~

4 ~~$\zeta = \texttt{map}[\alpha_1, \alpha_2]$~~

$$\begin{array}{lcl} \beta_1 & = & \beta_1 \\ \beta_2 & = & \delta \\ \alpha_1 & = & \delta \\ \alpha_2 & = & \delta \end{array}$$

Example (Continued)

gen : $\texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\beta_1] \to \texttt{map}[\alpha_1, \alpha_2]$

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{array}{ccc} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ *Type Error*

2 ~~$\gamma = \beta_1$~~

3 ~~$\texttt{map}[\beta_1, \beta_2] = \texttt{map}[\beta_1, \delta]$~~

3 ~~$\texttt{map}[\alpha_1, \alpha_2] = \texttt{map}[\delta, \delta]$~~

4 ~~$\xi = \texttt{map}[\alpha_1, \alpha_2]$~~

1 ~~$\beta_1 = \beta_1$~~

$$\begin{array}{ccc} \beta_2 & = & \delta \\ \alpha_1 & = & \delta \\ \alpha_2 & = & \delta \end{array}$$

Example (Continued)

gen : $\texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\beta_1] \rightarrow \texttt{map}[\alpha_1, \alpha_2]$

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{array}{lcl} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ *Type Error*

2 ~~$\gamma = \beta_1$~~

3 ~~$\texttt{map}[\beta_1, \beta_2] = \texttt{map}[\beta_1, \delta]$~~

3 ~~$\texttt{map}[\alpha_1, \alpha_2] = \texttt{map}[\delta, \delta]$~~

4 ~~$\xi = \texttt{map}[\alpha_1, \alpha_2]$~~

1 ~~$\beta_1 = \beta_1$~~

2 ~~$\beta_2 = \delta$~~

$\alpha_1 = \beta_2$

$\alpha_2 = \beta_2$

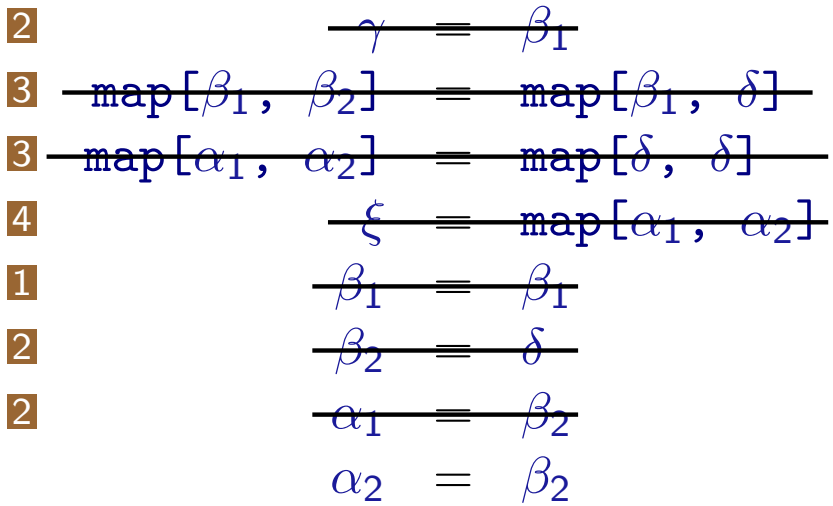
Example (Continued)

gen : $\texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\beta_1] \rightarrow \texttt{map}[\beta_2, \alpha_2]$

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{array}{rcl} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ *Type Error*



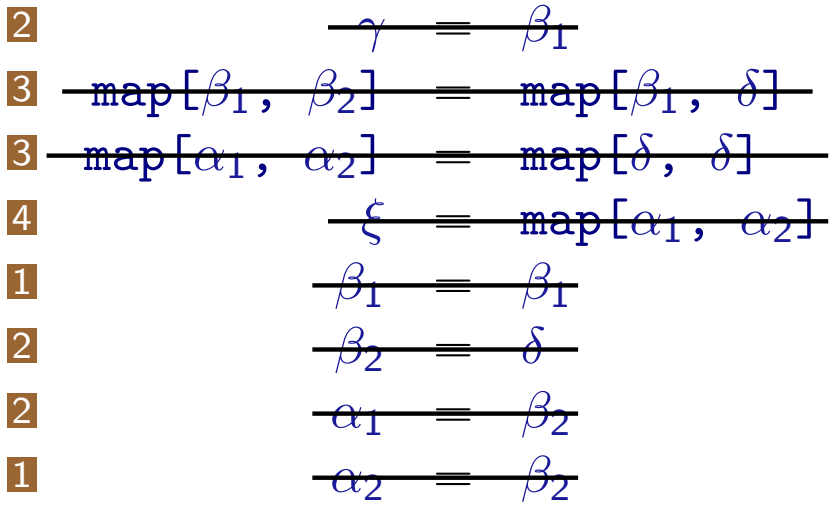
Example (Continued)

gen : $\texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\beta_1] \rightarrow \texttt{map}[\beta_2, \beta_2]$

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{array}{ccc} \tau_1^a & = & \tau_1^b \\ \ldots & & \ldots \\ \tau_k^a & = & \tau_k^b \end{array}$$

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ *Type Error*



Substituting "Everywhere"?

- The Martelli/Montanari algorithm asks us to "replace type variables everywhere":
  - "Solution: Replace β with α everywhere"
  - "Solution: Replace $C_i^k(\tau_1, \ldots, \tau_k)$ for α everywhere"
- Implementation strategies?:
  - **Substitute systematically**:
    - Replace everywhere in worklist
    - Replace everywhere in solutions (e.g., symbol table)
  - **Update Lists**:
    - 'Substitute systematically', but on demand, storing pending updates
  - **Stateful type variables**: (*my recommendation*)
    - Type variables remember their bindings, e.g. in $\Delta(\alpha)$
    - Some challenges with nontrivial merges

Summary

- During type analysis, we often encounter nontrivial equations over types
- To check these and extract relevant equalities, we use **Unification**
- The **Martelli/Montanari algorithm** is efficient for the types we have discussed so far
- Input:
  - A list of equations over types
- Output:
  - Bindings to type variables
  - Type variables such as α may be:
    - Replaced by a concrete type, such as $\textsc{Int}$
    - Replaced by another type variable, such as β
    - Replaced by a partially abstract type, such as $\textsc{List}[\gamma]$

Merging Variables

- Consider solving:

$$\begin{aligned} \alpha &= \beta \\ \beta &= \gamma \\ \gamma &= \delta \\ \delta &= \xi \end{aligned}$$

- Implementing unification with stateful variables naively can make it costly to figure out the "real" type of α:

$$\boxed{\Delta(\alpha) = \beta}$$
$$\boxed{\Delta(\beta) = \gamma}$$
$$\boxed{\Delta(\gamma) = \delta}$$
$$\boxed{\Delta(\delta) = \xi}$$

- Fast unification implementations instead use UNION-FIND datastructures

Union-Find Datastructures

Java

```
public class UFSet {
  UFSet repr = null;

  // Find & update representative
  public UFSet find() {
    UFSet r = this;
    while (r.repr != null) {
      r = r.repr;
    }
    this.repr = r;
    return r;
  }

  public void union(UFSet other) {
    other = other.find();
    UFSet r = this.find();
    // we can update r or other
    if (r != other) {
      other.repr = r;
  } }

  public boolean equals(UFSet o) {
    return this.find() == o.find();
} }
```

Summary

- UNION-FIND datastructure can speed up type variable merging
- Each UNION-FIND object represents a set of equivalent type variables
- Each set has one representative UNION-FIND object
- *find* operation finds that representative
  - caches most recent representative
- *union*(v_1, v_2) operation finds representatives r_1, r_2 of two variables
  - If $r_1 \neq r_2$, v_1, v_2 in different set
  - Then, update either representative of v_1 to now be v_2, or vice-versa
  - High-performance implementations make this decision based on:
    - set size
    - estimated "depth" of representative chains ('*rank*')

Unification, Types, and Re-use

Teal-0

```
fun id(x:α1):α2 = return x:α1;

var b:β1 := id("foo":string):β2
var c:γ1 := id(15:int):γ2
```

- What are the types here?
- We have $\alpha_1 = \alpha_2 = \texttt{string} = \texttt{int}$: type error!

Approach doesn't allow `id` on both `string` and `int`

Type Schemes

- We had the same issue before:

$$\frac{}{\text{nil} : \textsc{List}[\alpha]} \qquad \Longrightarrow \qquad \frac{\alpha \quad \textit{fresh}}{\text{nil} : \textsc{List}[\alpha]}$$

- We want a similar scheme for `id`: *create fresh type variables*
- However, we can't write custom rules for all user-defined functions!
- Polymorphism with user-defined functions:
  - *Type Schemes* (or *Polytypes*):
    (1) "normal" polymorphic type: $\alpha \to \alpha$
    (2) variables to replace by fresh ones: $\{ \alpha \}$
    short notation for (1)+(2): $\forall \alpha . \alpha \to \alpha$
  - `id` : $\forall \alpha . \alpha \to \alpha$
  - *Instantiate* type schemes with fresh type variables on demand:

```
          id: α2 → α2
var b := id("foo");
```

```
                id: α3 → α3
var b := id(255);
```

Using Type Schemes

- If we have a type scheme: *instantiate* scheme to use it
- Instantiating type schemes: (formalises of the last slide):

$$\frac{\Delta(x) = \forall \alpha_1, \ldots, \alpha_n.\tau \quad \beta_i \text{ fresh}, i \in \{1, \ldots, n\}}{x : \tau[\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n]} \ (\textit{t-var-inst})$$

- If we *want* a type scheme: *abstract* type into type scheme
- Abstracting type schemes:
  1. Infer type via unification: $f : \tau$
  2. Figure out which set of type variables to abstract: $\mathcal{T}$
  3. **Update** type schema: $\Delta(f) = \forall \mathcal{T}.\tau$

How do we find $\mathcal{T}$?

Summary

- **Polymorphic types** can take multiple forms
  - Contrast: **Monomorphic types** (e.g. INT)
  - We focus on *Parametric polymorphism* (Types with type parameters)
- We represent polymorphic types as type **Schemes**
  - Abstract over free type variables ($\forall$) to introduce schemes
- *Instantiate* schemes into types when referenced

Finding Type Schemes (1/3)

Teal-0

```
fun id(x) = return x;

var b = id("foo");
var c = id(17);
```

- When should we build the schema for `id`?
  - **After all unification:**
    Too late: have already run into type error (cf. earlier)
  - **Before all unification:**
    Too early:
    - Our first type constraint was: `id` : $\alpha_1 \to \alpha_2$
    - However, `id` : $\forall \alpha_1, \alpha_2 . \alpha_1 \to \alpha_2$ would be wrong:
      Tells us nothing about connection between α_1 and α_2
  - **During Unification:**
    - Must abstract *after* unifying all variables that "matter" to `id`
    - Must abstract *before* we use `id`'s type

Finding Type Schemes (2/3)

Teal-0

```
fun f(x0, x1, x2) = return g(x0 - 1, x1, x2);

fun g(y0, y1, y2) =
    if y0 == y1 {
        return y1;
    } else {
        return f(y1, y0, y2);
    }
```

- Can't build schema of `f` without analysing `g`
- Can't build schema of `g` without analysing `f`

***Mutual dependency*: Can't fully analyse one before the other**

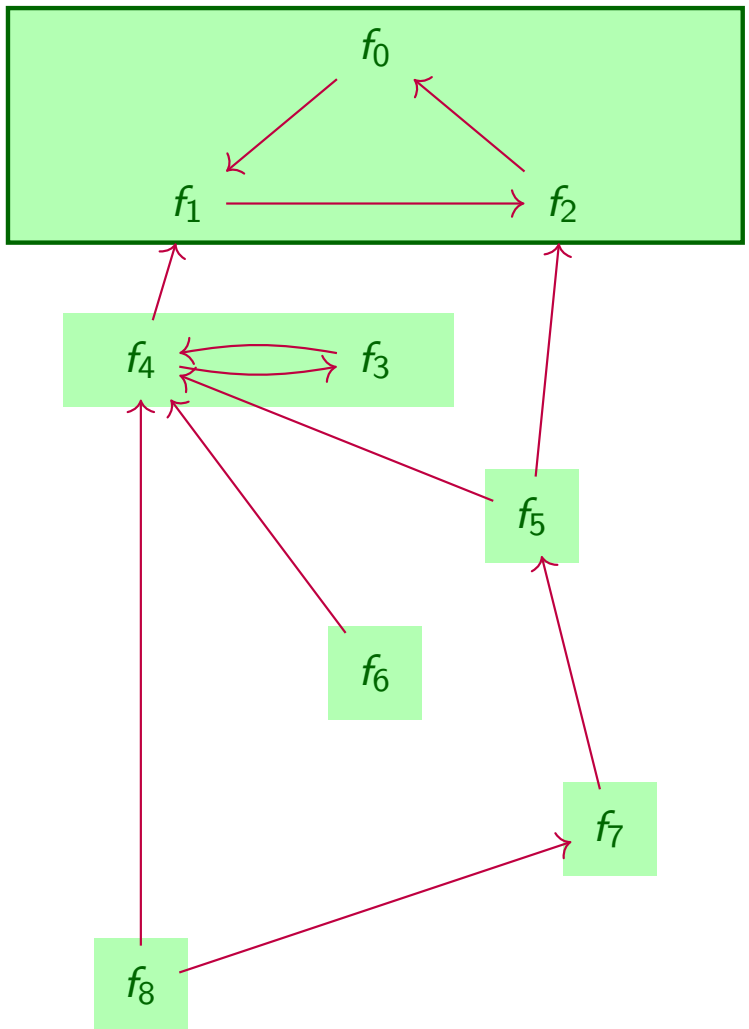
Finding Type Schemes (3/3)

- When functions call each other: must analyse them together
- Generalises to indirect calls
- Find *dependencies*:
  - if `f` calls `g`:
    - `f` *depends* (*directly*) on `g`
  - if `f` *depends* on `g` and `g` *depends* on `h`:
    - `f` *depends* on `h`
- `f` *depends* on `g`: Can't build schema for `f` before analysing `g`

$\Rightarrow$ Analyse such `f` and `g` together

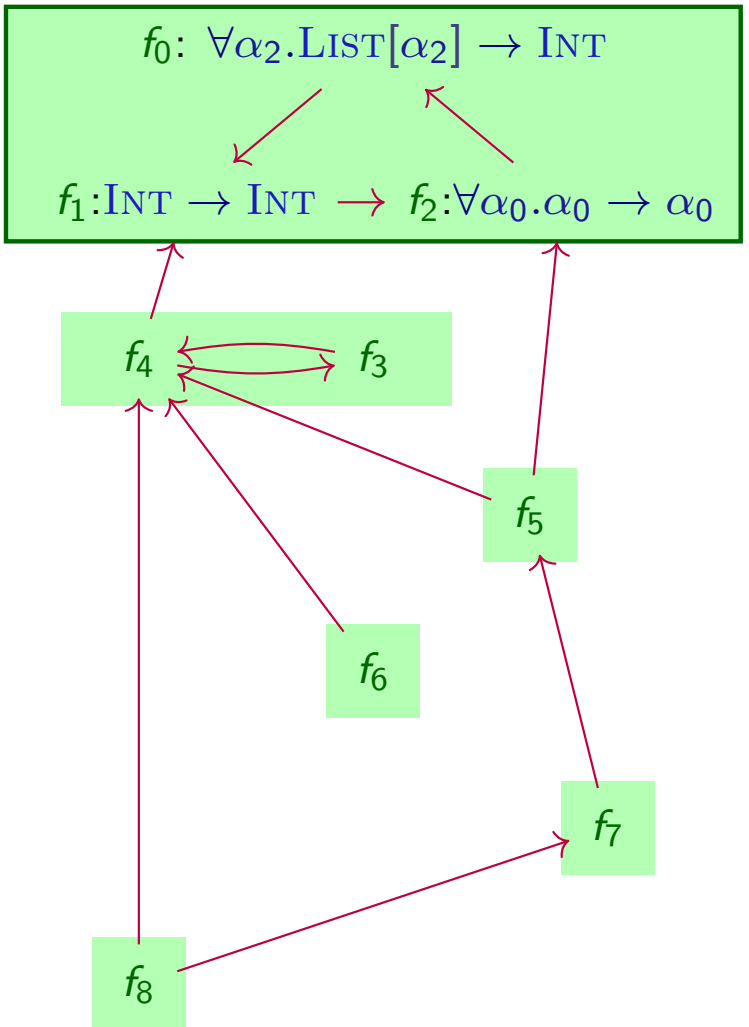
Polymorphic Type Inference Procedure

1. Determine dependencies
2. Cluster mutual dependencies
3. Mark clusters: untyped
4. While there are untyped clusters:
   5. Pick cluster that has no untyped dependencies



Polymorphic Type Inference Procedure

1. Determine dependencies
2. Cluster mutual dependencies
3. Mark clusters: untyped
4. While there are untyped clusters:
   5. Pick cluster that has no untyped dependencies
   6. Analyse all definitions in cluster:
      - Create fresh type variables as needed
      - Record typings: $\texttt{x} : \alpha$
      - Collect type equalities: $\tau_a = \tau_b$
   7. Run Unification
   8. For all definitions $\texttt{f} : \tau$ in cluster:
      - Let $\mathcal{T}$ = all type variables in τ
      - Set $\texttt{f} : \forall \mathcal{T}.\tau$



Summary

- **Polymorphic Type Inference**: generalise types with **Schemes**
- Algorithm:
  - Introduce type variables
  - Systematically apply typing rules to:
    - Generate typings
    - Generate type equality constraints
  - Unify equality constraints 'at the right time'
  - Abstract over free type variables ($\forall$) to introduce schemes
- Must analyse **Dependencies** between definitions
- Unify / abstract when:
  - Finished all dependencies
- Limitations:
  - Does not handle "inner functions"
    (See Damas-Hindley-Milner, Algorithms $\mathcal{W}$ / $\mathcal{J}$ if interested)
  - Type schemes over mutable objects (arrays etc.) unsound
  - Does not handle subtypes
    - Adding subtypes: see Dolan & Mycroft's "MLsub", 2017

Outlook

- Please consider moving your lab group to a later slot!
- Lab 0 still has lab priority for presenting this week
- Lab 1 is out
  - Lab 1a: priority next week
    - First half: constraint generation
    - Simple types
  - Lab 1b: priority in two weeks
    - Second half: constraint solving
    - All types

Next Week:

- Data Flow Analysis

`https://cs.lth.se/EDAP15`