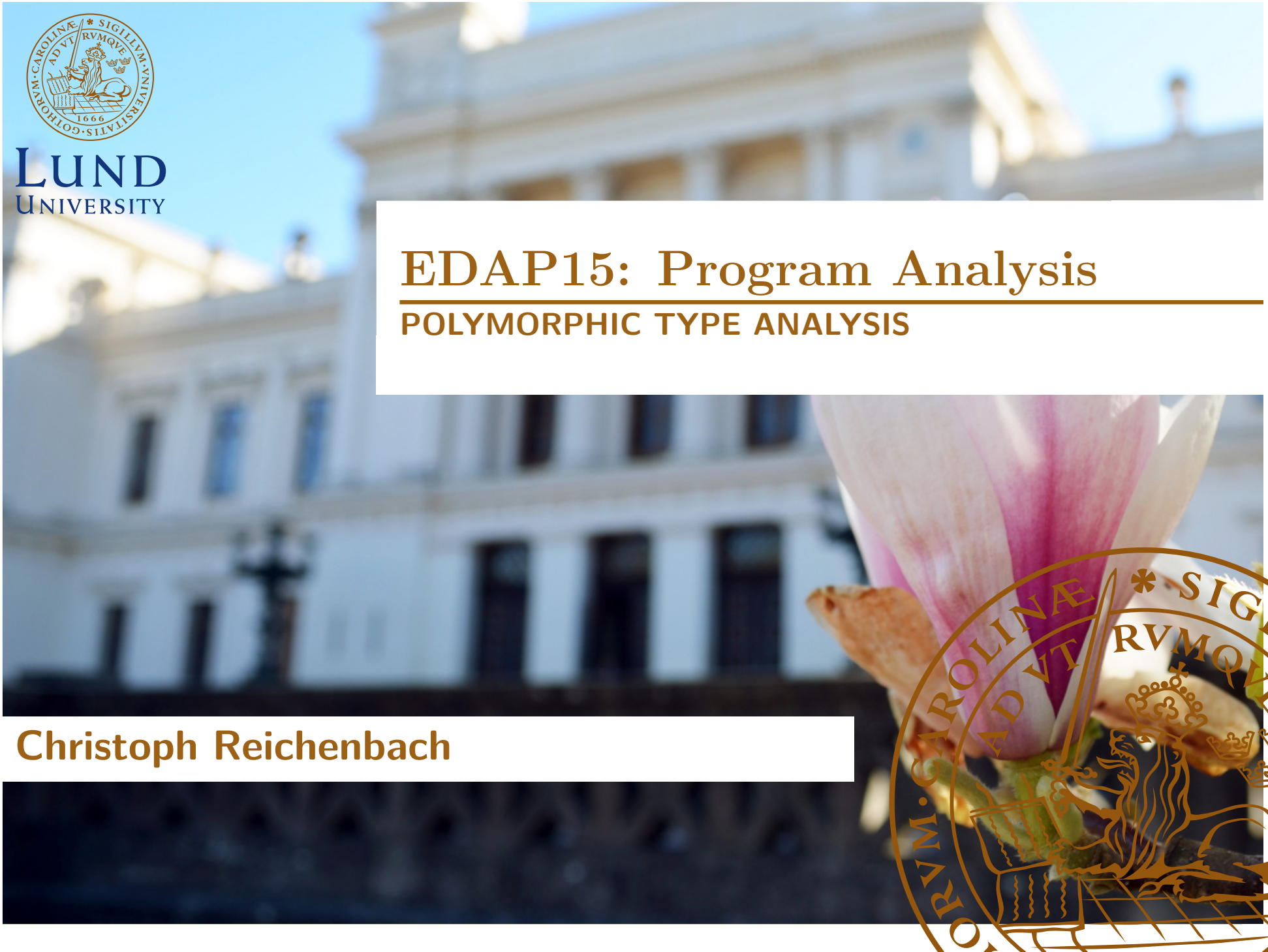


EDAP15: Program Analysis

POLYMORPHIC TYPE ANALYSIS

Christoph Reichenbach

Announcements

- Exercises start on Friday
- Call for Student Representative
- Video for second lecture damaged / unusable
  - Will record again later this semester
- Online office hours "on demand", please e-mail me

The Language LINGA

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \mathit{id} \\
 & | & \text{let } \mathit{id} = \langle \mathit{expr} \rangle \text{ in } \langle \mathit{expr} \rangle \\
 & | & \text{nil} \\
 & | & \text{cons } (\langle \mathit{expr} \rangle, \langle \mathit{expr} \rangle) \\
 & | & \langle \mathit{expr} \rangle \text{ plus } \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{expr} \rangle >= \langle \mathit{expr} \rangle \\
 & | & \text{if } \langle \mathit{expr} \rangle \text{ then } \langle \mathit{expr} \rangle \text{ else } \langle \mathit{expr} \rangle \\
\\
\mathit{val} & ::= & \mathit{nat} \\
 & | & \text{true} \ | \ \text{false} \\
\\
\mathit{ty} & ::= & \textsc{Int} \\
 & | & \textsc{Bool} \\
 & | & \textsc{List}\ [\langle \mathit{ty} \rangle] \\
 & | & \mathit{tyvar} \\
\\
\mathit{tyvar} & ::= & \alpha \ | \ \beta \ | \ \gamma \ | \ \ldots
\end{array}
$$

Parametric Types and Principal Types

- Problem last week:
  - $\mathtt{p} : \tau$ may have infinitely many τ, can't process all
  - One instance of more general problem:
    Having too many such τ makes analysis inefficient
- General approach: Find **Principal Type**
  - *Single* type that summarises all other types
- Here: use **Parametric Types** with **Type Variables**:
  - $\textsc{List}[\alpha]$ summarises $\textsc{List}[\textsc{Int}]$, $\textsc{List}[\textsc{Bool}]$, $\textsc{List}[\textsc{List}[\ldots]]$

Typing Rules for Parametric Types

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

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

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

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

Originally $\underline{x}.\texttt{ty} = \textsc{List}[\alpha]$

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

Analogous to variable types

$\underline{x}.\texttt{ty} = \cancel{\textsc{List}[\alpha]}\ \textsc{List}[\textsc{Int}]$

$\boxed{\alpha.\texttt{ty} = \textsc{Int}}$

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

Typing Rules for Parametric Types

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

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

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

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

Circular type — *t-cons* requires:

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

$$\boxed{\alpha.\texttt{ty} = \textsc{List}[\alpha]}$$

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

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

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

- Fix: We create a *fresh* type variable for every nil

$\boxed{\alpha_1\texttt{.ty} = (\textit{not set})}$

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

$\tau \mapsto \textsc{List}[\alpha_1]$

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

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:

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

More Uses for Type Variables

- Type variables help us defer decisions about types when we have no information
- Recall:

$$\frac{\underline{x}.\texttt{ty} = \tau}{\underline{x} : \tau}\ (\textit{t-var})$$

- This rule won't help us type e.g. function parameters:

Python

```
def f(x) :
    return (x, x)
```

- Can't apply *t-var* if we have never seen $\underline{x}$ before
- Instead, we can use a different rule for variables:

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

Summary

- We often need recursive types in our analyses
- As a result, some expressions may have an unbounded number of types
- We can usually use **type variables** to present these types practically
- This produces **principal types** if we can summarise *all* types
- **Parametric types** (or *parametrically polymorphic*) types arise frequently
- Correctly using expressions with type variables may require us to produce **fresh type variables**
- Open question:
  How *do* we merge type variables in equations?

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

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*:

$y : \tau$

Extract *equality constraints*:

$\tau_1 = \tau_2$

$$
\begin{array}{rl}
 & a : \texttt{map}[\beta_1, \beta_2] \\
 & b : \texttt{set}[\gamma] \\
 & \text{gen} : \texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\gamma] \rightarrow \xi \\
1 & m : \texttt{map}[\alpha_1, \alpha_2] \\
2 & v : \gamma \\
3 & v : \beta_1 \\
 & \gamma = \beta_1 \\
4 & x : \alpha_3 \\
 & a : \texttt{map}[\gamma, \alpha_3] \\
 & \texttt{map}[\beta_1, \beta_2] = \texttt{map}[\gamma, \alpha_3] \\
5 & m : \texttt{map}[\alpha_2, \beta_2] \\
 & \texttt{map}[\alpha_1, \alpha_2] = \texttt{map}[\alpha_2, \beta_2] \\
6 & m : \xi \\
 & \xi = \texttt{map}[\alpha_1, \alpha_2]
\end{array}
$$

How do we solve this automatically?

Type Inference: Constraints

Typings:

$$
\begin{array}{rcl}
\mathsf{a} & : & \texttt{map}[\beta_1,\ \beta_2] \\
\mathsf{b} & : & \texttt{set}[\gamma] \\
\mathsf{gen} & : & \texttt{map}[\beta_1,\ \beta_2] \times \texttt{set}[\gamma] \rightarrow \xi \\
\mathsf{m} & : & \texttt{map}[\alpha_1,\ \alpha_2] \\
\mathsf{v} & : & \gamma \\
\mathsf{v} & : & \beta_1 \\
\mathsf{x} & : & \alpha_3 \\
\mathsf{a} & : & \texttt{map}[\gamma,\ \alpha_3] \\
\mathsf{m} & : & \texttt{map}[\alpha_2,\ \beta_2] \\
\mathsf{m} & : & \xi
\end{array}
$$

Type Equality Constraints:

$$
\begin{array}{rcl}
\gamma & = & \beta_1 \\
\texttt{map}[\beta_1,\ \beta_2] & = & \texttt{map}[\gamma,\ \alpha_3] \\
\texttt{map}[\alpha_1,\ \alpha_2] & = & \texttt{map}[\alpha_2,\ \beta_2] \\
\xi & = & \texttt{map}[\alpha_1,\ \alpha_2]
\end{array}
$$

Unification

$$
\begin{aligned}
\gamma &= \beta_1 \\
\texttt{map}[\beta_1,\ \beta_2] &= \texttt{map}[\gamma,\ \alpha_3] \\
\texttt{map}[\alpha_1,\ \alpha_2] &= \texttt{map}[\alpha_2,\ \beta_2] \\
\xi &= \texttt{map}[\alpha_1,\ \alpha_2]
\end{aligned}
$$

- *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:
  - $\texttt{Set}[\alpha]$
  - $\texttt{Map}[\alpha, \beta]$
- Type constructors: things like $\texttt{Set}$, $\texttt{Map}$
  - 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.: $\texttt{int} \rightarrow \texttt{string} = C_{\rightarrow}^2(\texttt{int}, \texttt{string})$
  - E.g.: $Set[Set[\texttt{int}]] = C_{\texttt{Set}}^1(C_{\texttt{Set}}^1(\texttt{int}))$
- 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots) \Rightarrow \boxed{\textit{Type Error}}$ *($\leftarrow$ Occurs Check)*

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

Example (Continued)

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

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - $\boxed{\textit{Type Error}}$ 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ $\boxed{\textit{Type Error}}$

$$\begin{array}{rcl} \gamma & = & \beta_1 \\ \texttt{map}[\beta_1, \beta_2] & = & \texttt{map}[\gamma, \alpha_3] \\ \texttt{map}[\alpha_1, \alpha_2] & = & \texttt{map}[\alpha_2, \beta_2] \\ \xi & = & \texttt{map}[\alpha_1, \alpha_2] \end{array}$$

Example (Continued)

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

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - $\boxed{\textit{Type Error}}$ 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ $\boxed{\textit{Type Error}}$

2.

$$\begin{array}{rcl} \cancel{\gamma} & \cancel{=} & \cancel{\beta_1} \\ \texttt{map}[\beta_1, \beta_2] & = & \texttt{map}[\beta_1, \alpha_3] \\ \texttt{map}[\alpha_1, \alpha_2] & = & \texttt{map}[\alpha_2, \beta_2] \\ \xi & = & \texttt{map}[\alpha_1, \alpha_2] \end{array}$$

Example (Continued)

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

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - $\boxed{\textit{Type Error}}$ 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ $\boxed{\textit{Type Error}}$

$$\begin{array}{lrcl} 2 & \cancel{\gamma} & \cancel{=} & \cancel{\beta_1} \\ 3 & \cancel{\texttt{map}[\beta_1, \beta_2]} & \cancel{=} & \cancel{\texttt{map}[\beta_1, \alpha_3]} \\ & \texttt{map}[\alpha_1, \alpha_2] & = & \texttt{map}[\alpha_2, \beta_2] \\ & \xi & = & \texttt{map}[\alpha_1, \alpha_2] \\ & \beta_1 & = & \beta_1 \\ & \beta_2 & = & \alpha_2 \end{array}$$

Example (Continued)

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

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - $\boxed{\textit{Type Error}}$ 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
   $\Rightarrow$ $\boxed{\textit{Type Error}}$

$$\begin{array}{lrcl} 2 & \cancel{\gamma} & \cancel{=} & \cancel{\beta_1} \\ 3 & \cancel{\texttt{map}[\beta_1, \beta_2]} & \cancel{=} & \cancel{\texttt{map}[\beta_1, \alpha_3]} \\ 3 & \cancel{\texttt{map}[\alpha_1, \alpha_2]} & \cancel{=} & \cancel{\texttt{map}[\alpha_2, \beta_2]} \\ & \xi & = & \texttt{map}[\alpha_1, \alpha_2] \\ & \beta_1 & = & \beta_1 \\ & \beta_2 & = & \alpha_2 \\ & \alpha_1 & = & \alpha_2 \\ & \alpha_2 & = & \beta_2 \end{array}$$

Example (Continued)

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

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - $\boxed{\textit{Type Error}}$ 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ $\boxed{\textit{Type Error}}$

$$\begin{array}{lcl} 2 & \cancel{\gamma} & \cancel{=} & \cancel{\beta_1} \\ 3 & \cancel{\texttt{map}[\beta_1, \beta_2]} & \cancel{=} & \cancel{\texttt{map}[\beta_1, \alpha_3]} \\ 3 & \cancel{\texttt{map}[\alpha_1, \alpha_2]} & \cancel{=} & \cancel{\texttt{map}[\alpha_2, \beta_2]} \\ 4 & \cancel{\xi} & \cancel{=} & \cancel{\texttt{map}[\alpha_1, \alpha_2]} \\ & \beta_1 & = & \beta_1 \\ & \beta_2 & = & \alpha_2 \\ & \alpha_1 & = & \alpha_2 \\ & \alpha_2 & = & \beta_2 \end{array}$$

Example (Continued)

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

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - $\boxed{\textit{Type Error}}$ 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ $\boxed{\textit{Type Error}}$

$$\begin{array}{lcl} 2 & \cancel{\gamma} & \cancel{=} & \cancel{\beta_1} \\ 3 & \cancel{\texttt{map}[\beta_1, \beta_2]} & \cancel{=} & \cancel{\texttt{map}[\beta_1, \alpha_3]} \\ 3 & \cancel{\texttt{map}[\alpha_1, \alpha_2]} & \cancel{=} & \cancel{\texttt{map}[\alpha_2, \beta_2]} \\ 4 & \cancel{\xi} & \cancel{=} & \cancel{\texttt{map}[\alpha_1, \alpha_2]} \\ 1 & \cancel{\beta_1} & \cancel{=} & \cancel{\beta_1} \\ & \beta_2 & = & \alpha_2 \\ & \alpha_1 & = & \alpha_2 \\ & \alpha_2 & = & \beta_2 \end{array}$$

Example (Continued)

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

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - $\boxed{\textit{Type Error}}$ 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ $\boxed{\textit{Type Error}}$

$$\begin{array}{lcl} 2 & \cancel{\gamma} & \cancel{=} & \cancel{\beta_1} \\ 3 & \cancel{\texttt{map}[\beta_1, \beta_2]} & \cancel{=} & \cancel{\texttt{map}[\beta_1, \alpha_3]} \\ 3 & \cancel{\texttt{map}[\alpha_1, \alpha_2]} & \cancel{=} & \cancel{\texttt{map}[\alpha_2, \beta_2]} \\ 4 & \cancel{\xi} & \cancel{=} & \cancel{\texttt{map}[\alpha_1, \alpha_2]} \\ 1 & \cancel{\beta_1} & \cancel{=} & \cancel{\beta_1} \\ 2 & \cancel{\beta_2} & \cancel{=} & \cancel{\alpha_2} \\ & \alpha_1 & = & \beta_2 \\ & \beta_2 & = & \beta_2 \end{array}$$

Example (Continued)

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

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - $\boxed{\textit{Type Error}}$ 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
   $\Rightarrow$ $\boxed{\textit{Type Error}}$

$$\begin{array}{lrcl} 2 & \cancel{\gamma} & \cancel{=} & \cancel{\beta_1} \\ 3 & \cancel{\texttt{map}[\beta_1, \beta_2]} & \cancel{=} & \cancel{\texttt{map}[\beta_1, \alpha_3]} \\ 3 & \cancel{\texttt{map}[\alpha_1, \alpha_2]} & \cancel{=} & \cancel{\texttt{map}[\alpha_2, \beta_2]} \\ 4 & \cancel{\xi} & \cancel{=} & \cancel{\texttt{map}[\alpha_1, \alpha_2]} \\ 1 & \cancel{\beta_1} & \cancel{=} & \cancel{\beta_1} \\ 2 & \cancel{\beta_2} & \cancel{=} & \cancel{\alpha_2} \\ 2 & \cancel{\alpha_1} & \cancel{=} & \cancel{\beta_2} \\ & \alpha_1 & = & \alpha_1 \end{array}$$

Example (Continued)

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

1. $\alpha = \alpha$ (trivial)
2. $\alpha = \beta$
   - Replace $\beta \mapsto \alpha$
3. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - $\boxed{\textit{Type Error}}$ 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:** $\alpha = C_i^k(\ldots, \alpha, \ldots)$
     $\Rightarrow$ $\boxed{\textit{Type Error}}$

2	$\cancel{\gamma}$	$\cancel{=}$	$\cancel{\beta_1}$
3	$\cancel{\texttt{map}[\beta_1, \beta_2]}$	$\cancel{=}$	$\cancel{\texttt{map}[\beta_1, \alpha_3]}$
3	$\cancel{\texttt{map}[\alpha_1, \alpha_2]}$	$\cancel{=}$	$\cancel{\texttt{map}[\alpha_2, \beta_2]}$
4	$\cancel{\xi}$	$\cancel{=}$	$\cancel{\texttt{map}[\alpha_1, \alpha_2]}$
1	$\cancel{\beta_1}$	$\cancel{=}$	$\cancel{\beta_1}$
2	$\cancel{\beta_2}$	$\cancel{=}$	$\cancel{\alpha_2}$
2	$\cancel{\alpha_1}$	$\cancel{=}$	$\cancel{\beta_2}$
1	$\cancel{\alpha_1}$	$\cancel{=}$	$\cancel{\alpha_1}$

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 α`.ty`
    - 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:

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

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

$$\boxed{\alpha\,\texttt{.ty} = \beta}$$
$$\boxed{\beta\,\texttt{.ty} = \gamma}$$
$$\boxed{\gamma\,\texttt{.ty} = \delta}$$
$$\boxed{\delta\,\texttt{.ty} = \xi}$$

- Fast unification implementations instead use UNION-FIND datastructures

Union-Find Datastructures

Java

```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
- Type variables represent a set of equivalent variables
- Each set has one representative
- *find* operation finds that representative
  - updates cached references to it
- *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*')

Towards Real Languages: TEAL-0

```
module    ::= ⟨import⟩⋆ ⟨decl⟩⋆

import    ::= import ⟨qualified⟩ ;

qualified ::= id
            | ⟨qualified⟩ :: id

decl      ::= ⟨vardecl⟩ ;
            | fun id ( ⟨formals⟩? ) ⟨opttype⟩ = ⟨stmt⟩

vardecl   ::= var id ⟨opttype⟩
            | var id ⟨opttype⟩ := ⟨expr⟩;

formals   ::= id ⟨opttype⟩
            | id ⟨opttype⟩ , ⟨formal⟩

opttype   ::= : ⟨type⟩
            | ε

type      ::= int | string | any
            | array [ ⟨type⟩ ]

block     ::= { ⟨stmt⟩⋆ }
```

```
expr      ::= ⟨expr⟩ ⟨binop⟩ ⟨expr⟩
            | not ⟨expr⟩
            | ( ⟨expr⟩ ⟨opttype⟩ )
            | ⟨expr⟩ [ ⟨expr⟩ ]
            | id ( ⟨actuals⟩? )
            | [ ⟨actuals⟩? ]
            | new ⟨type⟩ ( ⟨expr⟩ )
            | int | string | null
            | id

actuals   ::= expr
            | expr, ⟨actuals⟩

binop     ::= + | - | * | / | %
            | == | != | < | <= | >= | >
            | or | and

stmt      ::= ⟨vardecl⟩
            | ⟨expr⟩ ;
            | ⟨expr⟩ := ⟨expr⟩ ;
            | ⟨block⟩
            | return ⟨expr⟩ ;
            | if ⟨expr⟩ ⟨block⟩ else ⟨block⟩
            | if ⟨expr⟩ ⟨block⟩
            | while ⟨expr⟩ ⟨block⟩
```

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!

`id` function doesn't work for both `string` and `int`!

Type Schemes

- We had the same issue before:

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

- 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 types:
  - *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: $\alpha_2 \to \alpha_2$

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

id: $\alpha_3 \to \alpha_3$

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

Using Type Schemes

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

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

- If we *want* a type scheme: *abstract* type into type scheme
- Abstracting type schemes:

1. Infer type via unification: $\underline{f} : \tau$
2. Figure out which set of type variables to abstract: $\mathcal{T}$
3. Assign type schema: $\underline{f}.\texttt{ty} = \forall \mathcal{T}.\tau$

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

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**

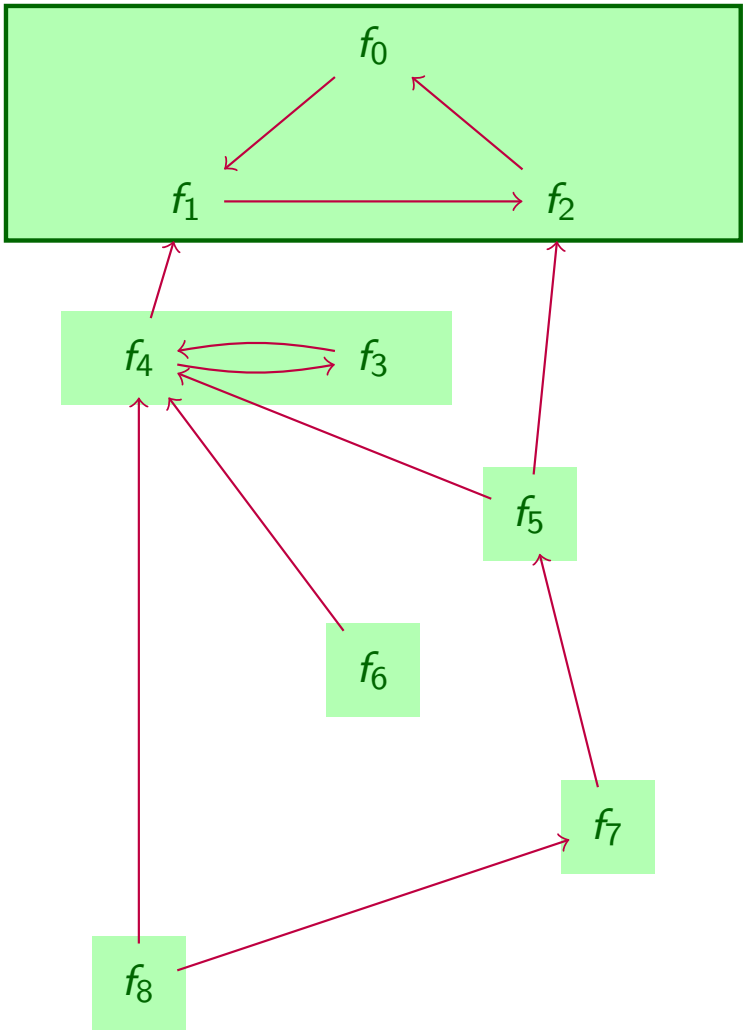
Finding Type Schemes (3/3)

- 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

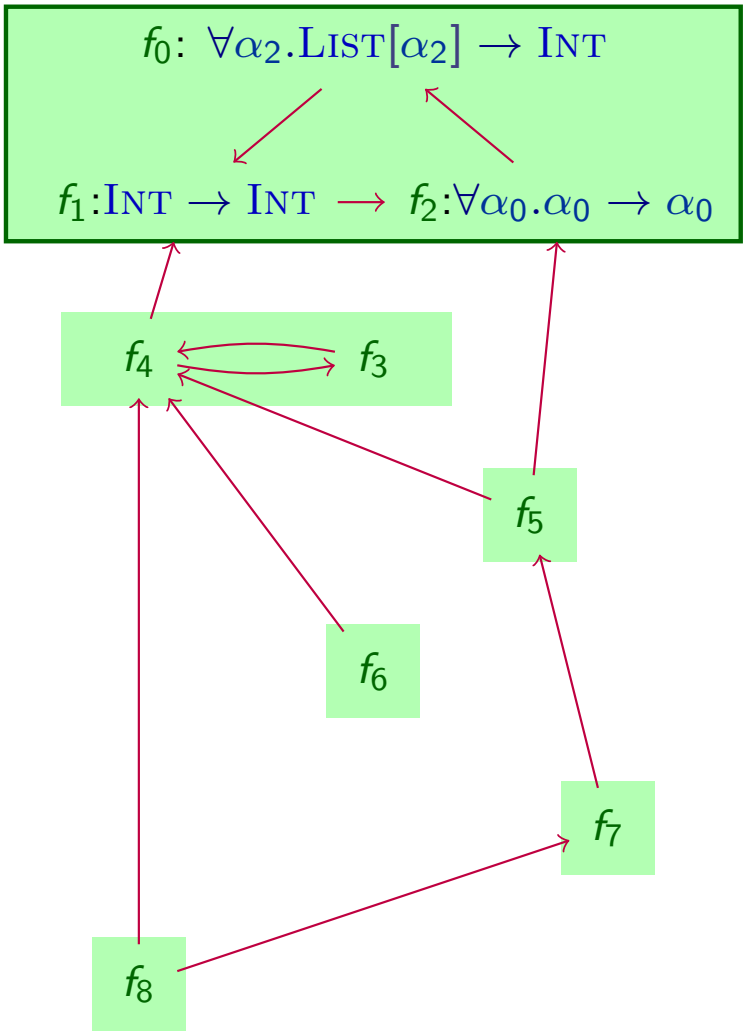
Polymorphic Type Inference Procedure

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



Summary

- **Polymorphic Type Inference** allows generalising 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
- The 'right time' to unify / abstract:
  - Have finished all dependencies?
  - Can unify all constraints within mutual dependency cluster
- Limitations:
  - Does not handle "inner functions"
    (See Damas-Hindley-Milner, Algorithms $\mathcal{W}$ / $\mathcal{J}$ if interested)
  - Does not handle subtypes

Review: Types for Program Analysis

$$\mathsf{p} : \tau$$

- Types summarise properties of programs
- Can summarise:
  - Computational results
  - Side effects
  - Dependencies

  ...
- *Type Analysis* processes programs recursively to
  - *Check* types
  - *Infer* types
- *Typing Rules* concisely express this process

$$\frac{e_1 : \textsc{Int} \quad e_2 : \textsc{Int}}{e_1 \ \mathsf{plus} \ e_2 : \textsc{Int}} \ (\textit{t-plus})$$

Parametric and Polymorphic Types

- *Polymorphism* in type analysis means that $\mathtt{p} : \tau$ may have many solutions for τ.
  - A *Principal Type* for p is a single type that summarises all τ.
- *Parametric Types* use *Type Variables* to form principal types
- Generic code creates a challenge to parametric types:

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

- *Type Schemes* make the type of f reusable:

$$\mathtt{f} : \forall \alpha . \alpha \rightarrow \alpha$$

  - Construct type schemes by *Abstraction*
  - *Instantiate* type schemes with *fresh type variables* on demand:

```
f(1)     :  int
f("x")   :  string
```

Example: Dependency Analysis (1/2)

Simplified TEAL-0:

$$
\begin{array}{lcl}
\mathit{module} & ::= & \langle \mathit{decl} \rangle \star \\
\mathit{decl} & ::= & \mathsf{fun}\ \mathit{id}\ (\ \mathit{id}\ )\ =\ \langle \mathit{stmt} \rangle \\
\mathit{expr} & ::= & \mathit{id}\ (\ \langle \mathit{expr} \rangle\ ) \\
 & \mid & \mathit{int} \\
 & \mid & \mathit{id} \\
\mathit{block} & ::= & \{\ \langle \mathit{stmt} \rangle \star\ \} \\
\mathit{stmt} & ::= & \mathsf{if}\ \langle \mathit{expr} \rangle\ \langle \mathit{block} \rangle\ \mathsf{else}\ \langle \mathit{block} \rangle \\
 & \mid & \mathsf{return}\ \langle \mathit{expr} \rangle\ ; \\
 & \mid & \langle \mathit{block} \rangle
\end{array}
$$

Goal: Let's build a dependency analysis

Example: Dependency Analysis (2/2)

- Type: *Set of direct dependencies* (invoked functions)

$$\frac{e : D}{\underline{f}(e) : D \cup \{\underline{f}\}} \qquad \frac{v \in int}{v : \emptyset} \qquad \frac{\underline{x} \in id}{\underline{x} : \emptyset}$$

$$\frac{e : D}{\textbf{return } e : D}$$

$$\frac{e : D_1 \quad b_1 : D_2 \quad b_2 : D_3}{\textbf{if } e \textbf{ then } b_2 \textbf{ else } b_3 : D_1 \cup D_2 \cup D_3}$$

$$\frac{s_i : D_i \text{ for all } i \in \{1, \ldots, n\}}{\{\ s_1 \ \ldots \ s_n\ \} : \bigcup_{i \in \{1,\ldots,n\}} D_i}$$

Building a Program Analysis