EDAP15: Program Analysis

POLYMORPHIC TYPE ANALYSIS

Christoph Reichenbach

LUND UNIVERSITY

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

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

Adding Lists: The Language LINGA

$$
\begin{array}{lcll}
\mathit{expr} & ::= & \langle \mathit{val} \rangle & \\
 & | & \mathit{id} & \\
 & | & \text{let } \mathit{id} = \langle \mathit{expr} \rangle \text{ in } \langle \mathit{expr} \rangle & \\
 & | & \text{nil} & \textbf{new!} \\
 & | & \text{cons } (\langle \mathit{expr} \rangle, \langle \mathit{expr} \rangle) & \textbf{new!} \\
 & | & \langle \mathit{expr} \rangle \text{ plus } \langle \mathit{expr} \rangle & \\
 & | & \langle \mathit{expr} \rangle >= \langle \mathit{expr} \rangle & \\
 & | & \text{if } \langle \mathit{expr} \rangle \text{ then } \langle \mathit{expr} \rangle \text{ else } \langle \mathit{expr} \rangle & \\
 & & & \\
\mathit{val} & ::= & \mathit{nat} & \\
 & | & \text{true} \mid \text{false} &
\end{array}
$$

- nil is the empty list
- cons(v, ℓ) takes list ℓ and prepends v
- Can express list [0, 1, 2] as:

cons(0, cons(1, cons(2, nil)))

The Type of Lists

The language of types $\mathbb{T}_{linga}$ has one new production:

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

Example types:

- cons(true, nil) : $\textsc{List}[\textsc{Bool}]$
- cons(1, cons(2, nil)) : $\textsc{List}[\textsc{Int}]$
- cons(1, cons(false, nil)) : *type error*
- cons(cons(1, nil), nil) : $\textsc{List}[\textsc{List}[\textsc{Int}]]$
- nil : $\textsc{List}[\textsc{Int}]$
  $\textsc{List}[\textsc{Bool}]$
  $\textsc{List}[\textsc{List}[\textsc{Int}]]$
  $\textsc{List}[\ldots, \textsc{List}[\textsc{Bool}], \ldots]$

First attempt at typing rules:

$$\frac{\tau \in \mathbb{T}_{linga}}{\text{nil} : \textsc{List}[\tau]} \ (\textit{t-nil})$$

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

nil has infinitely many of the types in $\mathbb{T}_{linga}$

Principal Types

- `p` : τ 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 subsumes all other possible
- Various approaches (for those who took EDAP05)?
  - *Parametric Polymorphism*, using parametric types
  - Subtype Polymorphism
  - Type Classes

  ...
- Here: use **Parametric Types** with **Type Variables**:
  - $\textsc{List}[\alpha]$ summarises $\textsc{List}[\textsc{Int}]$, $\textsc{List}[\textsc{Bool}]$, $\textsc{List}[\textsc{List}[\ldots]]$

Type Variables

$$
\begin{array}{lcl}
\textit{ty} & ::= & \textsc{Int} \\
 & | & \textsc{Bool} \\
 & | & \textsc{List}\ [\langle \textit{ty} \rangle] \\
 & | & \textit{tyvar} \\
\\
\textit{tyvar} & ::= & \alpha \ |\ \beta \ |\ \gamma \ |\ \ldots
\end{array}
$$

- Working with infinitely many types is impractical
- Summarise types by introducing **type variables** into $\mathbb{T}_{linga}$
- Can now define **parametric type** of nil:

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

Parametric Types can compactly summarise many possible types

Summary

- Precise analyses often need to know parameterise types with other types
  - $\text{LIST}[\text{LIST}[\text{INT}]]$
- Naïve *Recursive Types* are difficult to work with:
  - If we *don't* know a 'component type', we have potentially infinitely many types to remember
- **Parametric Types**:
  - $\text{LIST}[\alpha]$
  - Use **Type Variable** α to express that we know the type only *partially*
- **Principal Types**:
  - τ is *principal* for e if it *subsumes* all τ' with e : τ' (meaning of "subsumes" varies by type system/analysis)

Three Languages With Variables

Meta-Language

- Describes *Object Language*(s)
- Variables refer to object language concepts:
  - LINGA programs
  - $\mathbb{T}_{linga}$ types

Programs: LINGA

- "Object Language" #1
- Variables refer to input programs
- Example: $\underline{\text{x}}$ in

let $\underline{\text{x}}$ = 1 in $\underline{\text{x}}$

Types: $\mathbb{T}_{linga}$

- "Object Language" #2
- Variables refer to unknown types
- Example: α in

$\textsc{List}[\alpha]$

Meta-Variables Can Reference Object-Language Variables

Meta-Variable references	**Example**	**Meta-Variable Notation**
Program	1 plus 2	e
Type	$\textsc{List}[\textsc{Bool}]$	τ
Program variable	$\underline{\text{foo}}$	$\underline{x}$
Type Variable	α	α

Parametric Types and Principal Types

- Challenge:
  - p : τ 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

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

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

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

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

Originally $\Delta(\underline{x}) = \textsc{List}[\alpha]$
Must merge $\textsc{List}[\alpha] = \textsc{List}[\textsc{Int}]$
Analogous to variable types

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

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

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

Typing Rules for Parametric Types

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

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

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

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

Circular type — *t-cons* requires:

$$\tau = \textsc{List}[\alpha]$$
$$\textsc{List}[\tau] = \textsc{List}[\alpha]$$

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

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

Type Variable Freshness

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

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

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

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

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

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

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

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:

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

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?

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

More Uses for Type Variables

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

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

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

Python

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

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

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

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*:

$\tau_1 = \tau_2$

	$\mathsf{a} : \texttt{map}[\beta_1, \beta_2]$
	$\mathsf{b} : \texttt{set}[\gamma]$
	$\mathsf{gen} : \texttt{map}[\beta_1, \beta_2] \times \texttt{set}[\gamma] \to \xi$
1	$\mathsf{m} : \texttt{map}[\alpha_1, \alpha_2]$
2	$\mathsf{v} : \gamma$
3	$\mathsf{v} : \beta_1$
	$\gamma = \beta_1$
4	$\mathsf{x} : \alpha_3$
	$\mathsf{a} : \texttt{map}[\gamma, \alpha_3]$
	$\texttt{map}[\beta_1, \beta_2] = \texttt{map}[\gamma, \alpha_3]$
5	$\mathsf{m} : \texttt{map}[\alpha_3, \alpha_3]$
	$\texttt{map}[\alpha_1, \alpha_2] = \texttt{map}[\alpha_3, \alpha_3]$
6	$\mathsf{m} : \xi$
	$\xi = \texttt{map}[\alpha_1, \alpha_2]$

How do we solve this automatically?

Type Inference: Constraints

Typings:

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

Type Equality Constraints:

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

Unification

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

- *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`[α]
  - `Map`[α, β]
- 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.: $\texttt{int} \rightarrow \texttt{string} = C^2_{\rightarrow}(\texttt{int}, \texttt{string})$
  - E.g.: $Set[Set[\texttt{int}]] = C^1_{\texttt{Set}}(C^1_{\texttt{Set}}(\texttt{int}))$
- 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. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by equations:

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

4. $\alpha = C_i^k(\tau_1, \ldots, \tau_k)$
   - Solution: Replace $\alpha \mapsto C_i^k(\tau_1, \ldots, \tau_k)$ everywhere
   - **Except:** $\alpha = C_i^k(\ldots, \alpha, \ldots) \Rightarrow$ *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. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *Type Error* if $i \neq j$ or $k \neq l$
   - Otherwise: Replace by:

$$\begin{aligned} \tau_1^a &= \tau_1^b \\ \ldots & \quad \ldots \\ \tau_k^a &= \tau_k^b \end{aligned}$$

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*

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

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. $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$
   - *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

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

Example (Continued)

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 $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$

- *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*

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

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 $C_i^k(\tau_1^a, \ldots, \tau_k^a) = C_j^l(\tau_1^b, \ldots, \tau_l^b)$

- *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*

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

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 ~~$\texttt{map}[\beta_1, \beta_2] = \texttt{map}[\beta_1, \alpha_3]$~~

3 ~~$\texttt{map}[\alpha_1, \alpha_2] = \texttt{map}[\alpha_3, \alpha_3]$~~

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

$$\begin{array}{ccc} \beta_1 & = & \beta_1 \\ \beta_2 & = & \alpha_3 \\ \alpha_1 & = & \alpha_3 \\ \alpha_2 & = & \beta_2 \end{array}$$

Example (Continued)

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

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	~~γ~~	~~$=$~~	~~β_1~~
3	~~$\texttt{map}[\beta_1,\ \beta_2]$~~	~~$=$~~	~~$\texttt{map}[\beta_1,\ \alpha_3]$~~
3	~~$\texttt{map}[\alpha_1,\ \alpha_2]$~~	~~$=$~~	~~$\texttt{map}[\alpha_3,\ \alpha_3]$~~
4	~~ξ~~	~~$=$~~	~~$\texttt{map}[\alpha_1,\ \alpha_2]$~~
1	~~β_1~~	~~$=$~~	~~β_1~~
	β_2	$=$	α_3
	α_1	$=$	α_3
	α_2	$=$	β_2

Example (Continued)

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

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 ~~$\texttt{map}[\beta_1,\ \beta_2] = \texttt{map}[\beta_1,\ \alpha_3]$~~

3 ~~$\texttt{map}[\alpha_1,\ \alpha_2] = \texttt{map}[\alpha_3,\ \alpha_3]$~~

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

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

2 ~~$\beta_2 = \alpha_3$~~

$\alpha_1 = \beta_2$

$\alpha_2 = \beta_2$

Example (Continued)

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

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:

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

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	~~γ~~	~~$=$~~	~~β_1~~
3	~~$\texttt{map}[\beta_1, \beta_2]$~~	~~$=$~~	~~$\texttt{map}[\beta_1, \alpha_3]$~~
3	~~$\texttt{map}[\alpha_1, \alpha_2]$~~	~~$=$~~	~~$\texttt{map}[\alpha_3, \alpha_3]$~~
4	~~ξ~~	~~$=$~~	~~$\texttt{map}[\alpha_1, \alpha_2]$~~
1	~~β_1~~	~~$=$~~	~~β_1~~
2	~~β_2~~	~~$=$~~	~~α_3~~
2	~~α_1~~	~~$=$~~	~~β_2~~
	α_2	$=$	α_1

Example (Continued)

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

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:

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

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	~~γ~~	~~$=$~~	~~β_1~~
3	~~$\texttt{map}[\beta_1,\ \beta_2]$~~	~~$=$~~	~~$\texttt{map}[\beta_1,\ \alpha_3]$~~
3	~~$\texttt{map}[\alpha_1,\ \alpha_2]$~~	~~$=$~~	~~$\texttt{map}[\alpha_3,\ \alpha_3]$~~
4	~~ξ~~	~~$=$~~	~~$\texttt{map}[\alpha_1,\ \alpha_2]$~~
1	~~β_1~~	~~$=$~~	~~β_1~~
2	~~β_2~~	~~$=$~~	~~α_3~~
2	~~α_1~~	~~$=$~~	~~β_2~~
1	~~α_2~~	~~$=$~~	~~α_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 $\Delta(\alpha)$
    - 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
- 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: α2 → α2
var b := id("foo");
```

```
         id: α3 → α3
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):

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

- 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: $\Delta(\underline{f}) = \forall \mathcal{T}.\tau$

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

Summary

- Represent polymorphic types as type **Schemes**
- Abstract over free type variables ($\forall$) to introduce schemes
- *Instantiate* schemes into types when referenced

Outlook

- **Remember**:
  - Check for Videos and Quizzes tomorrow
- Next Lecture: Wednesday
  - Data Flow Analysis

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