



LUND
UNIVERSITY

EDAP15: Program Analysis

POLYMORPHIC TYPE ANALYSIS

Christoph Reichenbach



Lecture Overview

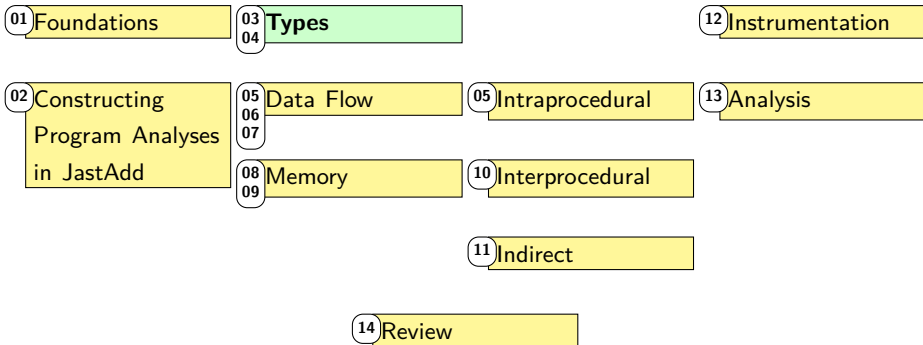
Foundations

Static Analysis

Dynamic
Analysis

Properties

Control Flow



Parametric Types and Principal Types

$$\begin{array}{lcl} Ty & ::= & \text{INT} \\ & | & \text{BOOL} \\ & | & \text{LIST } [\langle Ty \rangle] \end{array}$$

- ▶ Challenge:
 - ▶ $p : \tau$ allows infinitely many types
- ▶ To make practical: find **Principal Type**
 - ▶ *Single* type that summarises all other types
- ▶ Here: use **Parametric Types** with **Type Variables**:
 - ▶ $\text{LIST}[\alpha]$ summarises $\text{LIST}[\text{INT}]$, $\text{LIST}[\text{BOOL}]$, $\text{LIST}[\text{LIST}[\dots]]$
- ▶ Difference τ , α when computing type:
 - ▶ τ : must solve when we see AST node
 - ▶ α : is a valid type already
 - ▶ Can make it more concrete later

Typing Rules: Parametric Types

— First attempt —

$\frac{}{\text{true} : \text{BOOL}} \text{ (t-true)}$	$\frac{}{\text{false} : \text{BOOL}} \text{ (t-false)}$	$\frac{v \in \text{Nat}}{v : \text{INT}} \text{ (t-nat)}$
$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \text{ (t-plus)}$	$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \geq e_2 : \text{BOOL}} \text{ (t-ge)}$	$\frac{\Delta(\underline{x}) = \tau}{\underline{x} : \tau} \text{ (t-var)}$
$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{ (t-if)}$	$\frac{e_1 : \tau \quad \Delta(\underline{x}) = \tau_1 \quad e_2 : \tau_2}{\text{let } \underline{x} = e_1 \text{ in } e_2 : \tau_2} \text{ (t-let)}$	
$\frac{}{\text{nil} : \text{LIST}[\alpha]} \text{ (t-nil)}$	$\frac{e_1 : \tau \quad e_2 : \text{LIST}[\tau]}{\text{cons}(e_1, e_2) : \text{LIST}[\tau]} \text{ (t-cons)}$	

Originally $\Delta(\underline{x}) = \text{LIST}[\alpha]$
 Must merge $\text{LIST}[\alpha] = \text{LIST}[\text{INT}]$
 Analogous to variable types

$\Delta(\underline{x}) = \text{LIST}[\alpha] \quad \text{LIST}[\text{INT}]$

$\Delta(\alpha) = \text{INT}$

$\frac{}{\text{nil} : \text{LIST}[\alpha]} \text{ (t-nil)}$	$\frac{1 \in \text{Nat}}{1 : \text{INT}} \text{ (t-nat)}$	$\frac{\Delta(\underline{x}) = \text{LIST}[\text{INT}]}{\underline{x} : \text{LIST}[\text{INT}]}$
$\Delta(\underline{x}) = \text{LIST}[\alpha]$	$\text{cons}(1, \underline{x}) : \text{LIST}[\text{INT}]$	 (t-let)
$\text{let } \underline{x} = \text{nil} \text{ in } \text{cons}(1, \underline{x}) : \text{LIST}[\text{INT}]$		

Typing Rules: Parametric Types

— First attempt —

$$\frac{}{\text{true} : \text{BOOL}} \quad (t\text{-true})$$

$$\frac{}{\text{false} : \text{BOOL}} \quad (t\text{-false})$$

$$\frac{v \in \text{Nat}}{v : \text{INT}} \quad (t\text{-nat})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 \text{ plus } e_2 : \text{INT}} \quad (t\text{-plus})$$

$$\frac{e_1 : \text{INT} \quad e_2 : \text{INT}}{e_1 >= e_2 : \text{BOOL}} \quad (t\text{-ge})$$

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

$$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (t\text{-if})$$

$$\frac{e_1 : \tau \quad \Delta(x) = \tau_1 \quad e_2 : \tau_2}{\text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad (t\text{-let})$$

$$\frac{}{\text{nil} : \text{LIST}[\alpha]} \quad (t\text{-nil})$$

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

Circular type — $t\text{-cons}$ requires:

$$\tau = \text{LIST}[\alpha]$$

$$\text{LIST}[\tau] = \text{LIST}[\alpha]$$

$$\Delta(\alpha) = \text{LIST}[\alpha]$$

$$\frac{\text{nil} : \text{LIST}[\alpha] \quad \text{nil} : \text{LIST}[\alpha]}{\text{cons}(\text{nil}, \text{nil}) : ?} \quad \begin{matrix} (t\text{-nil}) \\ (t\text{-cons}) \end{matrix}$$

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

$$\frac{\alpha \text{ fresh}}{\text{nil} : \text{LIST}[\alpha]} \quad (t\text{-nil})$$

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

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

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

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

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

$$\frac{\frac{\alpha_1 \text{ fresh}}{\text{nil} : \tau \text{LIST}[\alpha_1]} \quad (t\text{-nil}) \quad \frac{\alpha_2 \text{ fresh}}{\text{nil} : \text{LIST}[\tau \text{LIST}[\alpha_1]] \text{LIST}[\alpha_2]} \quad (t\text{-nil})}{\text{cons}(\text{nil}, \text{nil}) : \text{LIST}[\text{LIST}[\alpha_1]]} \quad (t\text{-cons})$$

Summary

- ▶ Some expressions may have multiple or even unbounded number of types
 - ⇒ *Polymorphic expression*
- ▶ With **parametric types**: use **type variables** to present such types
 - ⇒ *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?

$$\text{LIST}[\alpha_1] = \text{LIST}[\text{LIST}[\alpha_2]]$$

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	$\alpha \rightarrow \beta$

- 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:

countOccurrencesInList : LIST[α] $\times$ $\alpha \rightarrow$ 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` $\rightarrow$... `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 [ $y$ ]
```

- ▶ $g: \dots \alpha \rightarrow \dots \text{LIST}[\alpha]$
- ▶ 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*?

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

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

⇒ no $\underline{\tau}$ exists for $\Delta(\underline{x}) = \underline{\tau}$

⇒ 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:

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

Typing Variables with Type Variables

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

Python

```
x = 1
```

$\Delta(\underline{x}) = \alpha$

$1 : \text{INT}$

$\alpha = \text{INT}$

```
x = "foo"
```

$\Delta(\underline{x}) = \beta$

$"\text{foo}" : \text{STRING}$

$\beta = \text{STRING}$

- ▶ Each assignment involves variable $\underline{x}$
- ▶ Create one type variable per use of $\underline{x}$
- ▶ Observe that $\alpha = \text{INT}$
- ▶ Observe that $\beta = \text{STRING}$
- ▶ Note that $\alpha = \Delta(\underline{x}) = \beta$:
Found type error
- ▶ Simplification: attach type variable to *declaration* of $\underline{x}$, so we only need α

Simplified use of Type Variables

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

Python

```
x = 1
```

$\underline{x} : \alpha$
 $1 : \text{INT}$
 $\alpha = \text{INT}$

```
x = "foo"
```

$\underline{x} : \alpha$
 $\text{"foo"} : \text{STRING}$
 $\alpha = \text{STRING}$

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*:

$y : \tau$

Extract *equality constraints*:

$\tau_1 = \tau_2$

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

How do we solve this automatically?

Summary

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

Type Inference: Constraints

Typings:

a : $\text{map}[\beta_1, \beta_2]$
b : $\text{set}[\gamma]$
gen : $\text{map}[\beta_1, \beta_2] \times \text{set}[\gamma] \rightarrow \xi$
m : $\text{map}[\alpha_1, \alpha_2]$
v : γ
v : β_1
x : α_3
a : $\text{map}[\gamma, \alpha_3]$
m : $\text{map}[\alpha_3, \alpha_3]$
m : ξ

Type Equality Constraints:

$\gamma = \beta_1$
 $\text{map}[\beta_1, \beta_2] = \text{map}[\gamma, \alpha_3]$
 $\text{map}[\alpha_1, \alpha_2] = \text{map}[\alpha_3, \alpha_3]$
 $\xi = \text{map}[\alpha_1, \alpha_2]$

Unification

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

- ▶ *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:
 - ▶ $\text{Set}[\alpha]$
 - ▶ $\text{Map}[\alpha, \beta]$
- ▶ Type constructors: things like Set , Map
 - ▶ Take type parameters α, β
 - ▶ Build new type
- ▶ Other type constructors:
 - ▶ $\dots \times \dots \times \dots$: constructs product types
 - ▶ $\rightarrow$: constructs function types
- ▶ General notation: $C_i^k(\tau_1, \dots, \tau_k)$
 - ▶ E.g.: $\text{int} \rightarrow \text{string} = C_{\rightarrow}^2(C_{\text{int}}^0, C_{\text{string}}^0)$
 - ▶ E.g.: $\text{Set}[\text{Set}[\text{int}]] = C_{\text{Set}}^1(C_{\text{Set}}^1(C_{\text{int}}^0))$
- ▶ 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 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

- Type Error if $i \neq j$ or $k \neq l$
- Otherwise: Replace by equations:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

- Solution: Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$ everywhere
- **Except:** $\alpha = C_i^k(\dots, \alpha, \dots) \Rightarrow$ Type Error (“Occurs Check”)

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

Example (Continued)

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

1 $\alpha = \alpha$ (trivial)

2 $\alpha = \beta$

► Replace $\beta \mapsto \alpha$

3 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

► **Type Error** if $i \neq j$ or $k \neq l$

► Otherwise: Replace by:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

► Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$

► **Except:** $\alpha = C_i^k(\dots, \alpha, \dots)$

$\Rightarrow$ **Type Error**

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

Example (Continued)

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

1 $\alpha = \alpha$ (trivial)

2 $\alpha = \beta$

► Replace $\beta \mapsto \alpha$

3 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

► **Type Error** if $i \neq j$ or $k \neq l$

► Otherwise: Replace by:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

► Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$

► **Except:** $\alpha = C_i^k(\dots, \alpha, \dots)$

⇒ **Type Error**

2

$$\alpha = \beta_1$$

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

$$\text{map}[\alpha_1, \alpha_2] = \text{map}[\alpha_3, \alpha_3]$$

$$\xi = \text{map}[\alpha_1, \alpha_2]$$

Example (Continued)

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

1 $\alpha = \alpha$ (trivial)

2 $\alpha = \beta$

► Replace $\beta \mapsto \alpha$

3 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

► **Type Error** if $i \neq j$ or $k \neq l$

► Otherwise: Replace by:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

► Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$

► **Except:** $\alpha = C_i^k(\dots, \alpha, \dots)$

$\Rightarrow$ **Type Error**

2

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

3

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

$$\text{map}[\alpha_1, \alpha_2] = \text{map}[\alpha_3, \alpha_3]$$

$$\xi = \text{map}[\alpha_1, \alpha_2]$$

$$\beta_1 = \beta_1$$

$$\beta_2 = \alpha_3$$

Example (Continued)

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

1 $\alpha = \alpha$ (trivial)

2 $\alpha = \beta$

► Replace $\beta \mapsto \alpha$

3 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

► Type Error if $i \neq j$ or $k \neq l$

► Otherwise: Replace by:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

► Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$

► **Except:** $\alpha = C_i^k(\dots, \alpha, \dots)$

⇒ Type Error

2

~~$\gamma = \beta_1$~~

3

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

3

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

$\xi = \text{map}[\alpha_1, \alpha_2]$

$\beta_1 = \beta_1$

$\beta_2 = \alpha_3$

$\alpha_1 = \alpha_3$

$\alpha_2 = \alpha_3$

Example (Continued)

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

1 $\alpha = \alpha$ (trivial)

2 $\alpha = \beta$

▶ Replace $\beta \mapsto \alpha$

3 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

▶ **Type Error** if $i \neq j$ or $k \neq l$

▶ Otherwise: Replace by:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

▶ Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$

▶ **Except:** $\alpha = C_i^k(\dots, \alpha, \dots)$

$\Rightarrow$ **Type Error**

2

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

3

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

3

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

4

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

$$\beta_1 = \beta_1$$

$$\beta_2 = \alpha_3$$

$$\alpha_1 = \alpha_3$$

$$\alpha_2 = \alpha_3$$

Example (Continued)

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

1 $\alpha = \alpha$ (trivial)

2 $\alpha = \beta$

► Replace $\beta \mapsto \alpha$

3 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

► Type Error if $i \neq j$ or $k \neq l$

► Otherwise: Replace by:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

► Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$

► **Except:** $\alpha = C_i^k(\dots, \alpha, \dots)$

⇒ Type Error

2

~~$\gamma = \beta_1$~~

3

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

3

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

4

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

1

~~$\beta_1 = \beta_1$~~

$\beta_2 = \alpha_3$

$\alpha_1 = \alpha_3$

$\alpha_2 = \alpha_3$

Example (Continued)

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

1 $\alpha = \alpha$ (trivial)

2 $\alpha = \beta$

► Replace $\beta \mapsto \alpha$

3 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

► **Type Error** if $i \neq j$ or $k \neq l$

► Otherwise: Replace by:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

► Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$

► **Except:** $\alpha = C_i^k(\dots, \alpha, \dots)$

⇒ **Type Error**

2

~~$\gamma = \beta_1$~~

3

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

3

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

4

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

1

~~$\beta_1 = \beta_1$~~

2

~~$\beta_2 = \alpha_3$~~

$\alpha_1 = \beta_2$

$\alpha_2 = \beta_2$

Example (Continued)

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

1 $\alpha = \alpha$ (trivial)

2 $\alpha = \beta$

► Replace $\beta \mapsto \alpha$

3 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

► Type Error if $i \neq j$ or $k \neq l$

► Otherwise: Replace by:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

► Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$

► **Except:** $\alpha = C_i^k(\dots, \alpha, \dots)$

⇒ Type Error

2

~~$\gamma = \beta_1$~~

3

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

3

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

4

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

1

~~$\beta_1 = \beta_1$~~

2

~~$\beta_2 = \alpha_3$~~

2

~~$\alpha_1 = \beta_2$~~

$\alpha_2 = \beta_2$

Example (Continued)

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

1 $\alpha = \alpha$ (trivial)

2 $\alpha = \beta$

► Replace $\beta \mapsto \alpha$

3 $C_i^k(\tau_1^a, \dots, \tau_k^a) = C_j^l(\tau_1^b, \dots, \tau_l^b)$

► **Type Error** if $i \neq j$ or $k \neq l$

► Otherwise: Replace by:

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

4 $\alpha = C_i^k(\tau_1, \dots, \tau_k)$

► Replace $\alpha \mapsto C_i^k(\tau_1, \dots, \tau_k)$

► **Except:** $\alpha = C_i^k(\dots, \alpha, \dots)$

⇒ **Type Error**

2

~~$\gamma = \beta_1$~~

3

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

3

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

4

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

1

~~$\beta_1 = \beta_1$~~

2

~~$\beta_2 = \alpha_3$~~

2

~~$\alpha_1 = \beta_2$~~

1

~~$\alpha_2 = \beta_2$~~

Substituting “Everywhere”?

- ▶ The Martelli/Montanari algorithm asks us to “replace type variables everywhere”:
 - ▶ “Solution: Replace β with α everywhere”
 - ▶ “Solution: Replace $C_i^k(\tau_1, \dots, \tau_k)$ 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 `INT`
 - ▶ Replaced by another type variable, such as β
 - ▶ Replaced by a partially abstract type, such as `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 :

$\Delta(\alpha) = \beta$
$\Delta(\beta) = \gamma$
$\Delta(\gamma) = \delta$
$\Delta(\delta) = \xi$

- 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’*)

Unification, Types, and Re-use

Teal-0

```
fun id(x: $\alpha_1$ ): $\alpha_2$  = return x: $\alpha_1$ ;  
var b: $\beta_1$  := id("foo":string): $\beta_2$   
var c: $\gamma_1$  := id(15:int): $\gamma_2$ 
```

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

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

Type Schemes

- ▶ We had the same issue before:

$$\frac{}{\text{nil} : \text{LIST}[\alpha]} \Longrightarrow \frac{\alpha \text{ fresh}}{\text{nil} : \text{LIST}[\alpha]}$$

- ▶ 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: $\alpha \rightarrow \alpha$
 - (2) variables to replace by fresh ones: $\{ \alpha \}$short notation for (1)+(2): $\forall \alpha. \alpha \rightarrow \alpha$
 - ▶ `id`: $\forall \alpha. \alpha \rightarrow \alpha$
 - ▶ *Instantiate* type schemes with fresh type variables on demand:

id: $\alpha_2 \rightarrow \alpha_2$
`var b := id("foo");`

id: $\alpha_3 \rightarrow \alpha_3$
`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):

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

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

- 1 Infer type via unification: $\underline{f} : \underline{\tau}$
- 2 Figure out which set of type variables to abstract: $\mathcal{T}$
- 3 **Update** type schema: $\Delta(\underline{f}) = \forall \mathcal{T}. \underline{\tau}$

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

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: $\text{id} : \alpha_1 \rightarrow \alpha_2$
 - ▶ However, $\text{id} : \forall \alpha_1, \alpha_2. \alpha_1 \rightarrow \alpha_2$ 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`

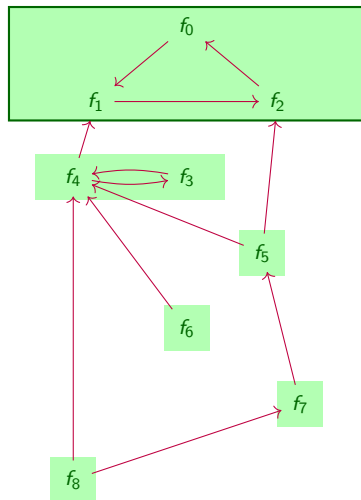
Mutual dependency: Can't fully analyse one before the other

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
- ⇒ Analyse such f and g together

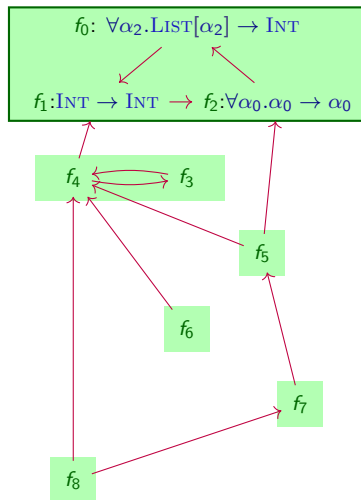
Polymorphic Type Inference Procedure

- 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: $x : \alpha$
 - ▶ Collect type equalities: $\tau_a = \tau_b$
 - 7 Run Unification
 - 8 For all definitions $f : \tau$ in cluster:
 - ▶ Let $\mathcal{T}$ = all type variables in τ
 - ▶ Set $f : \forall \mathcal{T}. \tau$



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

- ▶ Lab 0 still has lab priority for presenting this week
- ▶ Lab 1 will be released today
 - ▶ 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

<http://cs.lth.se/EDAP15>