

# TYPE INFERENCE (I)

# RESPONSE TO CRITICISMS OF TYPED LANGUAGES

- Types overly constrain functions & data
  - **Polymorphism** makes typed constructs useful in more contexts
    - universal polymorphism => code reuse
      - $\lambda x.x : 'a \rightarrow 'a$  (\* 'a is any type \*)
      - $\text{reverse} : 'a \text{ list} \rightarrow 'a \text{ list}$  (\* 'a is any type \*)
    - existential polymorphism => modules & abstract data types
      - $T = \exists X \{a: X; f: X \rightarrow \text{bool}\}$
      - $\text{intT} = \{a: \text{int}; f: \text{int} \rightarrow \text{bool}\}$
      - $\text{boolT} = \{a: \text{bool}; f: \text{bool} \rightarrow \text{bool}\}$
- Types clutter programs and slow down programmer productivity
  - **Type inference.**
    - uninformative annotations may be omitted

# TYPE SCHEMES

- A **type scheme** contains type variables that may be filled in during type inference
  - $s ::= 'a \mid \text{int} \mid \text{bool} \mid s1 \rightarrow s2$
  - 'a is a type variable (which stands for  $\alpha$ )
- A **term scheme** is a term (a.k.a. expression) that contains type schemes rather than proper types
  - $e ::= \dots \mid \text{fun } f(x:s1) : s2 = e$
  - Note the above *named function* notation

# UNTYPED LANGUAGE

- $e ::=$

$x$

|  $c$  (consts: 0, 1, ..., true, false)

|  $e_1 \text{ bop } e_2$  (binary operations)

|  $\text{fun } f(x) = e$  (named function, can be recursive)

|  $e_1 e_2$  (applications)

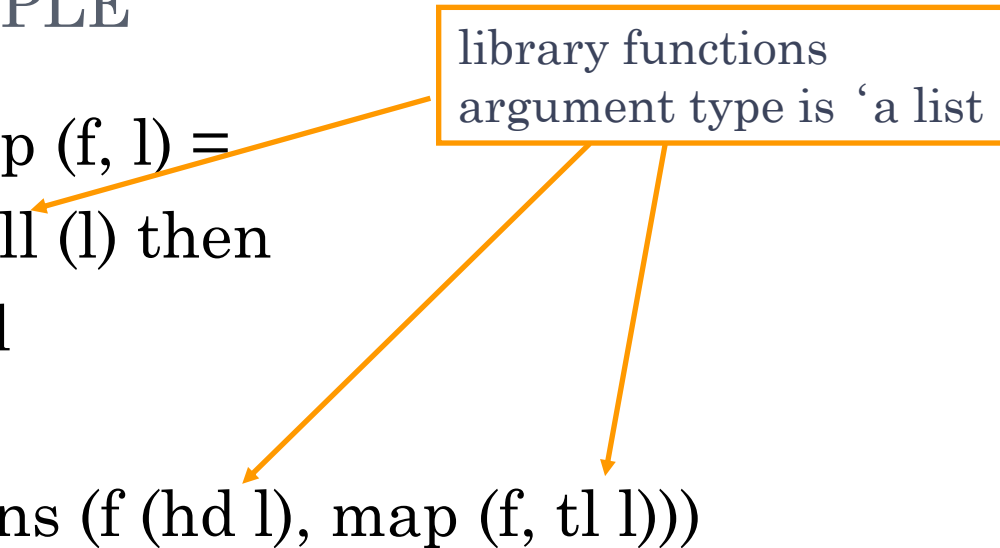
## EXAMPLE

```
fun map (f, l) =  
  if null (l) then  
    nil  
  else  
    cons (f (hd l), map (f, tl l)))
```

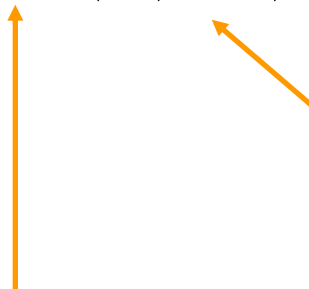
# EXAMPLE

```
fun map (f, l) =  
  if null (l) then  
    nil  
  else  
    cons (f (hd l), map (f, tl l))
```

library functions  
argument type is 'a list



library function  
argument type is ('a \*  
'a list)  
result type is 'a list



result type is 'a



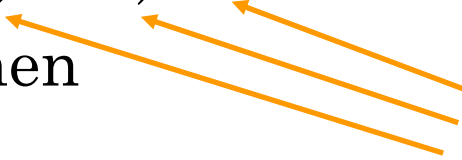
result type is 'a list



## STEP 1: ADD TYPE SCHEMES

```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil  
  else  
    cons (f (hd l), map (f, tl l)))
```

type schemes  
on functions



## STEP 2: GENERATE CONSTRAINTS


```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil  
  else  
    cons (f (hd l), map (f, tl l)))
```

- walk over the program & keep track of the type equations  $t1 = t2$  that must hold in order to type check the expressions according to the normal typing rules
- introduce new type variables for unknown types whenever necessary



## STEP 2: GENERATE CONSTRAINTS

```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil  
  else  
    cons (f (hd l), map (f, tl l)))
```

 **b = b' list**

## STEP 2: GENERATE CONSTRAINTS

constraints  
b = b' list

```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil : d list  
  else  
    cons (f (hd l), map (f, tl l)))
```

## STEP 2: GENERATE CONSTRAINTS

constraints  
b = b' list

```
fun map (f : a, l : b) : c =
```

```
  if null (l) then
```

```
    nil : d list
```

```
  else
```

```
    cons (f (hd l), map (f, tl l)))
```

b = b'' list



b = b''' list



## STEP 2: GENERATE CONSTRAINTS

constraints  
b = b' list

```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil : d list  
  else  
    cons (f (hd l), map (f, tl l: b''' list)))
```

b = b'' list

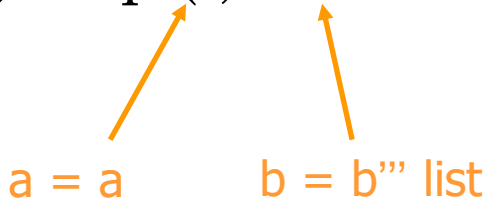


b = b''' list



## STEP 2: GENERATE CONSTRAINTS

```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil : d list  
  else  
    cons (f (hd l : b"), map (f, tl l: b"" list)))
```

  
a = a      b = b"" list

constraints

b = b' list

b = b"" list

b = b"" list

## STEP 2: GENERATE CONSTRAINTS

```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil : d list  
  else  
    cons (f (hd l : b'') : a', map (f, tl l) : c))
```

$a = b'' \rightarrow a'$



constraints

$b = b'$  list

$b = b''$  list

$b = b'''$  list

$a = a$

$b = b'''$  list

## STEP 2: GENERATE CONSTRAINTS


```
fun map (f : a, l : b) : c =
```

```
  if null (l) then
```

```
    nil : d list
```

```
  else
```

```
    cons (f (hd l) : a', map (f, tl l) : c) : c' list
```

  
c = c' list  
a' = c'

constraints

b = b' list

b = b'' list

b = b''' list

a = a

b = b''' list

a = b'' -> a'

## STEP 2: GENERATE CONSTRAINTS

```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil : d list  
  else  
    cons (f (hd l), map (f, tl l)) : c' list
```

d list = c' list



constraints

b = b' list

b = b'' list

b = b''' list

a = a

b = b''' list

a = b'' -> a'

c = c' list

a' = c'



## STEP 2: GENERATE CONSTRAINTS

```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil  
  else  
    cons (f (hd l), map (f, tl l))
```

: d list

d list = c

constraints  
b = b' list  
b = b'' list  
b = b''' list  
a = a  
b = b''' list  
a = b'' -> a'  
c = c' list  
a' = c'  
d list = c' list

## STEP 2: GENERATE CONSTRAINTS

```
fun map (f : a, l : b) : c =  
  if null (l) then  
    nil  
  else  
    cons (f (hd l), map (f, tl l)))
```

```
final  
b = b' list  
b = b'' list  
b = b''' list  
a = a  
b = b''' list  
a = b'' -> a'  
c = c' list  
a' = c'  
d list = c' list  
d list = c
```

## STEP 3: SOLVE CONSTRAINTS

- Constraint solution provides all possible solutions to type scheme annotations on terms

final  
constraints  
 $b = b'$  list  
 $b = b''$  list  
 $b = b'''$  list  
 $a = a$   
...



solution  
 $a = b' \rightarrow c'$   
 $b = b'$  list  
 $c = c'$  list



$\text{map } (f : b' \rightarrow c'$   
     $x : b' \text{ list})$   
:  $c'$  list  
=  
...

## STEP 4: GENERATE TYPES

- Generate types from type schemes
  - Option 1: pick **an instance** of the most general type when we have completed type inference on the entire program
    - $\text{map} : ((\text{int} \rightarrow \text{int}) * \text{int list}) \rightarrow \text{int list}$
  - Option 2: generate polymorphic types for program parts and continue (polymorphic) type inference
    - $\text{map} : \forall(a,b) ((a \rightarrow b) * \text{a list}) \rightarrow \text{b list}$

## QUIZ: GENERATING TYPES

Generate the polymorphic types for the following function:

```
fun fold (f, a, l) =  
  case l of  
    nil => a  
  | h::t => fold (f, f (h, a), t)
```

*Please show the intermedia steps and the equations that you are solving.*

# TYPE INFERENCE DETAILS

- **Type constraints** are sets of equations between type schemes
  - $q ::= \{s_{11} = s_{12}, \dots, s_{n1} = s_{n2}\}$
  - eg:  $\{b = b' \text{ list}, a = b \rightarrow c\}$

# CONSTRAINT GENERATION

- **Syntax-directed** constraint generation
  - our algorithm crawls over abstract syntax of untyped expressions and generates
    - a term scheme
    - a set of constraints
- Algorithm defined as set of inference rules (as always).
- Judgement form:
  - $G \dashv\vdash u \implies e : t, q$
  - $u$  is untyped expression
  - $e : t$  is a term scheme
  - $q$  is a set of constraints

# CONSTRAINT GENERATION

## ○ Simple rules:

- $G \dashv\vdash x \implies x : s, \{\}$  (if  $G(x) = s$ )
  - If  $G(x)$  is not defined then  $x$  is free variable
- $G \dashv\vdash 3 \implies 3 : \text{int}, \{\}$  (same for other ints)
- $G \dashv\vdash \text{true} \implies \text{true} : \text{bool}, \{\}$
- $G \dashv\vdash \text{false} \implies \text{false} : \text{bool}, \{\}$



# OPERATORS

$$\begin{array}{l} G \vdash\!\!-\! u_1 \implies e_1 : t_1, q_1 \qquad G \vdash\!\!-\! u_2 \implies e_2 : t_2, q_2 \\ \hline G \vdash\!\!-\! u_1 + u_2 \implies e_1 + e_2 : \text{int}, q_1 \cup q_2 \cup \{t_1 = \text{int}, t_2 = \text{int}\} \end{array}$$
$$\begin{array}{l} G \vdash\!\!-\! u_1 \implies e_1 : t_1, q_1 \qquad G \vdash\!\!-\! u_2 \implies e_2 : t_2, q_2 \\ \hline G \vdash\!\!-\! u_1 < u_2 \implies e_1 < e_2 : \text{bool}, q_1 \cup q_2 \cup \{t_1 = \text{int}, t_2 = \text{int}\} \end{array}$$

# IF STATEMENTS

$G \vdash u1 \implies e1 : t1, q1$

$G \vdash u2 \implies e2 : t2, q2$

$G \vdash u3 \implies e3 : t3, q3$

---

$G \vdash \text{if } u1 \text{ then } u2 \text{ else } u3 \implies \text{if } e1 \text{ then } e2 \text{ else } e3: a,$   
 $q1 \cup q2 \cup q3 \cup \{t1 = \text{bool}, a = t2, a = t3\}$

# FUNCTION APPLICATION

$$G \vdash u1 \implies e1 : t1, q1$$
$$G \vdash u2 \implies e2 : t2, q2$$

---

$$G \vdash u1 \ u2 \implies e1 \ e2 : a, q1 \cup q2 \cup \{t1 = t2 \rightarrow a\}$$

# FUNCTION DECLARATION

$$G, f : a \rightarrow b, x : a \dashv\vdash u \implies e : t, q$$

---

$$G \dashv\vdash \text{fun } f(x) = u \implies \text{fun } f(x : a) : b = e$$
$$: a \rightarrow b, q \cup \{t = b\}$$

(a, b are fresh type variables; not in G)

# SOLVING CONSTRAINTS

- A **solution** to a system of type constraints is a **substitution  $S$** 
  - a **function** from *type variables* to *type schemes*
  - substitutions are defined on all type variables (a total function), but only some of the variables are actually changed:
    - $S(a) = a$  (for almost all variables  $a$ )
    - $S(a) = s$  (for some  $a$  and some type scheme  $s$ )
  - $\text{dom}(S) = \text{set of variables s.t. } S(a) \neq a$

# SUBSTITUTIONS

- Given a substitution  $S$ , we can define a function  $S^*$  from *type schemes* (as opposed to type variables) to *type schemes*:
  - $S^*(\text{int}) = \text{int}$
  - $S^*(s1 \rightarrow s2) = S^*(s1) \rightarrow S^*(s2)$
  - $S^*(a) = S(a)$
- For simplicity, next I will write  $S(s)$  instead of  $S^*(s)$
- $s$  denotes type schemes, whereas  $a, b, c$  denote type variables
- This function replaces all type variables in a type scheme.

# COMPOSITION OF SUBSTITUTIONS

- **Composition** ( $U \circ S$ ) applies the substitution  $S$  and then applies the substitution  $U$ :
  - $(U \circ S)(a) = U(S(a))$
- We will need to compare substitutions
  - $T \leq S$  if  $T$  is “less general” than  $S$
  - $T \leq S$  if  $T$  is “more specific” than  $S$
  - Formally:  $T \leq S$  if and only if  $T = U \circ S$  for some  $U$

# COMPOSITION OF SUBSTITUTIONS

## ○ Examples:

- example 1: any substitution is less general than the identity substitution I:
  - $S \leq I$  because  $S = S \circ I$
- example 2:
  - $S(a) = \text{int}, S(b) = c \rightarrow c$
  - $T(a) = \text{int}, T(b) = c \rightarrow c, T(c) = \text{int}$
  - we conclude:  $T \leq S$
  - if  $T(a) = \text{int}, T(b) = \text{int} \rightarrow \text{bool}$  then T is unrelated to S (neither more nor less general)



# SOLVING A CONSTRAINT

- Judgment format:  $S \models q$   
( $S$  is a solution to the constraints  $q$ )

$$\frac{}{S \models \{ \}}$$



any substitution is  
a solution for the empty  
set of constraints

$$\frac{S(s1) = S(s2) \quad S \models q}{S \models \{s1 = s2\} \cup q}$$



a solution to an equation  
is a substitution that makes  
left and right sides equal

# MOST GENERAL SOLUTIONS

- S is the **principal** (most general) solution of a set of constraints  $q$  if
  - $S \models q$  (S is a solution)
  - if  $T \models q$  then  $T \leq S$  (S is the most general one)
- **Lemma:** If  $q$  has a solution, then it has a most general one
- We care about principal solutions since they will give us the most general types for terms (polymorphism!)
- **Exercise:**  
Prove: If  $q$  has a solution, then it has a most general one.

# EXAMPLES

## ○ Example 1

- $q = \{a=int, b=a\}$
- principal solution  $S$ :
  - $S(a) = S(b) = int$
  - $S(c) = c$  (for all  $c$  other than  $a, b$ )

# EXAMPLES

## ○ Example 2

- $q = \{a=\text{int}, b=a, b=\text{bool}\}$
- principal solution S:
  - does not exist (there is no solution to  $q$ )