Type Inference

COS 326
David Walker
Princeton University
The type for map looks like this:

\[
\text{map} : (\forall a, b. \, (a \to b) \to a \text{ list} \to b \text{ list})
\]

This type includes an implicit quantifier at the outermost level. So really, map’s type is this one:

\[
\text{map} : \forall a, b. \, (a \to b) \to a \text{ list} \to b \text{ list}
\]

To use a value with type \(\forall a, b, c \cdot t\), we first substitute types for parameters \(a, b, c\). eg:

\[
\text{map} \, (\text{fun} \, x \to x + 1) \, [2;3;4]
\]

here, we substitute \([\text{int}/a][\text{int}/b]\) in map’s type and then use map at type \((\text{int} \to \text{int}) \to \text{int list} \to \text{int list}\)
Type Checking (Simple Types)

A function check : context -> exp -> type

• requires function arguments to be annotated with types

• specified using formal rules. eg, the rule for function call:

\[
\frac{G |- e_1 : t_1 \rightarrow t_2 \quad G |- e_2 : t_1}{G |- e_1 e_2 : t_2}
\]
A **type scheme** contains type variables that may be filled in during type inference

\[
s ::= a \mid \text{int} \mid \text{bool} \mid s \rightarrow s
\]

A **term scheme** is a term that contains type schemes rather than proper types. eg, for functions:

\[
\text{fun (x:s) } \rightarrow \text{e}
\]

\[
\text{let rec f (x:s) : s = e}
\]
Two Algorithms for Inferring Types

Algorithm 1:
• Declarative; generates constraints to be solved later
• Easier to understand
• Easier to prove correct
• Less efficient, not used in practice

Algorithm 2:
• Imperative; solves constraints and updates as-you-go
• Harder to understand
• Harder to prove correct
• More efficient, used in practice
• See: http://okmij.org/ftp/ML/generalization.html
Algorithm 1

1) Add distinct variables in all places type schemes are needed

2) Generate constraints (equations between types) that must be satisfied in order for an expression to type check
   • Notice the difference between this and the type checking algorithm from last time. Last time, we tried to:
     • eagerly deduce the concrete type when checking every expression
     • reject programs when types didn't match. eg:
       \[ f : e \quad \text{-- f's argument type must equal e} \]
   • This time, we'll collect up equations like:
     \[ (a \rightarrow b) = c \]

3) Solve the equations, generating substitutions of types for var's
Example: Inferring types for map

```ocaml
let rec map f l =
  match l with
  | []    -> []
  | hd::tl -> f hd :: map f tl
```

let rec map (f:a) (l:b) : c =
match l with
  [] -> []
| (hd:d)::(tl:g) ->
  f hd :: map f tl
Step 2: Generate Constraints

let rec map (f:a) (l:b) : c =
  match l with
  [] -> []
| (hd:d)::(tl:g) ->
  f hd :: map f tl

b = d list
a = d -> e
...
Step 2: Generate Constraints

let rec map (f:a) (l:b) : c =
  match l with
  [] -> []
  | hd::tl -> f hd :: map f tl

final constraints:

b = b' list
b = b'' list
b = b''' list
a = a
a = b'' -> a'
c = c' list
c' = c'
d list = c' list
d list = c
Step 3: Solve Constraints

```ocaml
let rec map (f:a) (l:b) : c =
  match l with
  [] -> []
| hd::tl -> f hd :: map f tl
```

final 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`
- `d list = c`

final solution:

- `[b' -> c'/a]`
- `[b' list/b]`
- `[c' list/c]`
Step 3: Solve Constraints

let rec map (f:a) (l:b) : c =
    match l with
    [] -> []
    | hd::tl -> f hd :: map f tl

final solution:
[b' -> c'/a]
[b' list/b]
[c' list/c]

let rec map (f:b' -> c') (l:b' list) : c' list =
    match l with
    [] -> []
    | hd::tl -> f hd :: map f tl
Step 3: Solve Constraints

```plaintext
let rec map (f:a) (l:b) : c =
  match l with
  | [] -> []
  | hd::tl -> f hd :: map f tl
renaming type variables:

let rec map (f: 'a -> 'b) (l: 'a list): 'b list =
  match l with
  | [] -> []
  | hd::tl -> f hd :: map f tl
```
Type inference details

Type constraints are sets of equations between type schemes

- $q ::= \{s_{11} = s_{12}, ..., s_{n1} = s_{n2}\}$

- e.g.: $\{b = b' \text{ list, } a = (b -> c)\}$
Syntax-directed constraint generation

– our algorithm crawls over abstract syntax of untyped expressions and generates
  • a term scheme
  • a set of constraints
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:

- \( G \vdash u \Rightarrow e : t, q \)

Constraints that must be solved

- context
- unannotated expression
- annotated expression
- type (scheme)
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:
– $G \vdash u \Rightarrow e : t, q$

constraints that must be solved

context

unannotated expression

annotated expression

type (scheme)

in OCaml:

```ocaml
gen : ctxt -> exp ->
     ann_exp * scheme * constraints
```
Constraint Generation

Simple rules:

- $G \vdash x \Rightarrow x : s, \{\}$ (if $G(x) = s$)

- $G \vdash 3 \Rightarrow 3 : \text{int}, \{\}$ (same for other ints)

- $G \vdash \text{true} \Rightarrow \text{true} : \text{bool}, \{\}$

- $G \vdash \text{false} \Rightarrow \text{false} : \text{bool}, \{\}$
Operators

\[
G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \quad G \vdash u_2 \Rightarrow e_2 : t_2, q_2
\]

\[
\begin{align*}
G \vdash u_1 + u_2 \Rightarrow e_1 + e_2 : \text{int}, q_1 \cup q_2 \cup \{t_1 = \text{int}, t_2 = \text{int}\}
\end{align*}
\]

\[
G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \quad G \vdash u_2 \Rightarrow e_2 : t_2, q_2
\]

\[
\begin{align*}
G \vdash u_1 < u_2 \Rightarrow e_1 < e_2 : \text{bool}, q_1 \cup q_2 \cup \{t_1 = \text{int}, t_2 = \text{int}\}
\end{align*}
\]
If statements

\[
G \vdash u_1 \implies e_1 : t_1, q_1 \\
G \vdash u_2 \implies e_2 : t_2, q_2 \\
G \vdash u_3 \implies e_3 : t_3, q_3
\]

-----------------------------------------------------------------

\[
G \vdash \text{if } u_1 \text{ then } u_2 \text{ else } u_3 \implies \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t_2, \quad q_1 \cup q_2 \cup q_3 \cup \{t_1=\text{bool}, t_2 = t_3\}
\]
Function Application

\[
\begin{align*}
& G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \\
& G \vdash u_2 \Rightarrow e_2 : t_2, q_2 \\
& \text{(for fresh } a) \\
\hline
& G \vdash u_1 \ u_2 \Rightarrow e_1 \ e_2 : a, \ q_1 \ U \ q_2 \ U \ \{t_1 = t_2 \rightarrow a\}
\end{align*}
\]
Function Declaration

\[ G, x : a \vdash u \Rightarrow e : t, q \quad \text{(for fresh } a) \]

\[ G \vdash \text{fun } x \rightarrow u \Rightarrow \text{fun } (x : a) \rightarrow e : a \rightarrow t, q \]
Function Declaration

\[ G, f : a \rightarrow b, x : a \vdash u \Rightarrow e : t, q \quad (\text{for fresh } a,b) \]

\[ \frac{}{G \vdash \text{rec } f(x) = u \Rightarrow \text{rec } f(x : a) : b = e \quad : \quad a \rightarrow b, q \cup \{t = b\}} \]
Summary: The type inference system

G ⊢ u₁ ==> e₁ : t₁, q₁  
G ⊢ u₂ ==> e₂ : t₂, q₂  
----------------------------------------------------------------------------------  
G ⊢ u₁ + u₂ ==> e₁ + e₂ : int, q₁ U q₂ U {t₁ = int, t₂ = int}

G ⊢ x ==> x : s,  { }  (if G(x) = s)  
G ⊢ 3 ==> 3 : int, { }

G ⊢ u₁ ==> e₁ : t₁, q₁  
G ⊢ u₂ ==> e₂ : t₂, q₂  
G ⊢ u₃ ==> e₃ : t₃, q₃  
----------------------------------------------------------------------------------  
G ⊢ if u₁ then u₂ else u₃ ==> if e₁ then e₂ else e₃  : t₂, q₁ U q₂ U q₃ U {t₁=bool, t₂ = t₃}

G ⊢ u₁ ==> e₁ : t₁, q₁  
G ⊢ u₂ ==> e₂ : t₂, q₂  
G ⊢ u₃ ==> e₃ : t₃, q₃  
----------------------------------------------------------------------------------  
G ⊢ if u₁ then u₂ else u₃ ==> if e₁ then e₂ else e₃  : t₂, q₁ U q₂ U q₃ U {t₁=bool, t₂ = t₃}

G ⊢ if u₁ then u₂ else u₃ ==> if e₁ then e₂ else e₃  : t₂, q₁ U q₂ U q₃ U {t₁=bool, t₂ = t₃}

G ⊢ x ==> x : s,  { }  (if G(x) = s)  
G ⊢ 3 ==> 3 : int, { }

G ⊢ u₁ ==> e₁ : t₁, q₁  
G ⊢ u₂ ==> e₂ : t₂, q₂  
G ⊢ u₃ ==> e₃ : t₃, q₃  
----------------------------------------------------------------------------------  
G ⊢ u₁ u₂ ==> e₁ e₂  : a, q₁ U q₂ U {t₁ = t₂ -> a}

G, x : a ⊢ u ==> e : t, q  
----------------------------------------------------------------------------------  
G ⊢ fun x → u ==> fun (x : a) → e  : a → t, q

G, f : a → b, x : a ⊢ u ==> e : t, q  
----------------------------------------------------------------------------------  
G ⊢ rec f(x) = u ==> rec f (x : a) : b = e  : a → b, q U {t = b}
A solution to a system of type constraints is a *substitution S*

- a function from type variables to types
- assume substitutions are defined on all type variables:
  - $S(a) = a$ (for almost all variables $a$)
  - $S(a) = s$ (for some type scheme $s$)
- $\text{dom}(S) = \text{set of variables s.t. } S(a) \neq a$
A solution to a system of type constraints is a *substitution* $S$

- a function from type variables to type schemes
- assume substitutions are defined on all type variables:
  
  - $S(a) = a$  (for almost all variables $a$)
  - $S(a) = s$  (for some type scheme $s$)
- $\text{dom}(S) = \text{set of variables s.t. } S(a) \neq a$

We can also apply a substitution $S$ to a full type scheme $s$.

apply:  [ int/a,  int->bool/b ]

to:  b -> a -> b

returns: (int->bool) -> int -> (int->bool)
Substitutions

When is a substitution S a solution to a set of constraints?

Constraints:  \{ s1 = s2, s3 = s4, s5 = s6, ... \}

When the substitution makes both sides of all equations the same.

Eg:

constraints:
\[
\begin{align*}
a &= b \rightarrow c \\
c &= \text{int} \rightarrow \text{bool}
\end{align*}
\]
When is a substitution $S$ a solution to a set of constraints?

Constraints: $\{ s_1 = s_2, s_3 = s_4, s_5 = s_6, \ldots \}$

When the substitution makes both sides of all equations the same.

Eg:

Constraints:
- $a = b \rightarrow c$
- $c = \text{int} \rightarrow \text{bool}$

Solution:
- $b \rightarrow (\text{int} \rightarrow \text{bool})/a$
- $\text{int} \rightarrow \text{bool}/c$
- $b/b$
When is a substitution $S$ a solution to a set of constraints?

Constraints: $\{ s_1 = s_2, s_3 = s_4, s_5 = s_6, \ldots \}$

When the substitution makes both sides of all equations the same.

Eg:

- **constraints:**
  
  \[
  \begin{align*}
  a &= b \to c \\
  c &= \text{int} \to \text{bool}
  \end{align*}
  \]

- **solution:**
  
  \[
  \begin{align*}
  b \to (\text{int} \to \text{bool}) &= a \\
  \text{int} \to \text{bool} &= \text{int} \to \text{bool} \\
  b &= b
  \end{align*}
  \]

- **constraints with solution applied:**
  
  \[
  \begin{align*}
  b \to (\text{int} \to \text{bool}) &= b \to (\text{int} \to \text{bool}) \\
  \text{int} \to \text{bool} &= \text{int} \to \text{bool}
  \end{align*}
  \]
Substitutions

When is a substitution \( S \) a solution to a set of constraints?

Constraints: \( \{ s_1 = s_2, s_3 = s_4, s_5 = s_6, \ldots \} \)

When the substitution makes both sides of all equations the same.

A second solution

constraints:

\[
\begin{align*}
a &= b \rightarrow c \\
c &= \text{int} \rightarrow \text{bool}
\end{align*}
\]

solution 1:

\[
\begin{align*}
b &\rightarrow (\text{int} \rightarrow \text{bool}) \ / \ a \\
\text{int} \rightarrow \text{bool} &/ \ c \\
b &/ \ b
\end{align*}
\]

solution 2:

\[
\begin{align*}
\text{int} &\rightarrow (\text{int} \rightarrow \text{bool}) \ / \ a \\
\text{int} \rightarrow \text{bool} &/ \ c \\
\text{int} &/ \ b
\end{align*}
\]
Substitutions

When is one solution better than another to a set of constraints?

constraints:  
\[
\begin{align*}
a &= b \rightarrow c \\
c &= \text{int} \rightarrow \text{bool}
\end{align*}
\]

solution 1:  
\[
\begin{align*}
b &\rightarrow (\text{int} \rightarrow \text{bool}) \ / \ a \\
\text{int} \rightarrow \text{bool} &\ / \ c \\
b &\ / \ b
\end{align*}
\]

solution 2:  
\[
\begin{align*}
\text{int} \rightarrow (\text{int} \rightarrow \text{bool}) &\ / \ a \\
\text{int} \rightarrow \text{bool} &\ / \ c \\
\text{int} &\ / \ b
\end{align*}
\]

type \( b \rightarrow c \) with solution applied:
\[
\begin{align*}
b &\rightarrow (\text{int} \rightarrow \text{bool})
\end{align*}
\]

type \( b \rightarrow c \) with solution applied:
\[
\begin{align*}
\text{int} &\rightarrow (\text{int} \rightarrow \text{bool})
\end{align*}
\]
Solution 1 is "more general" – there is more flex.
Solution 2 is "more concrete"
We prefer solution 1.
Solution 1 is "more general" – there is more flex.
Solution 2 is "more concrete"
We prefer the more general (less concrete) solution 1.
Technically, we prefer T to S if there exists another substitution U and for all types t, S (t) = U (T (t))
Substitutions

There is always a **best** solution, which we can a principal solution. The best solution is (at least as) preferred as any other solution.

solution 1:

\[ b \rightarrow (\text{int} \rightarrow \text{bool})/a \]
\[ \text{int} \rightarrow \text{bool}/c \]
\[ b/b \]

solution 2:

\[ \text{int} \rightarrow (\text{int} \rightarrow \text{bool})/a \]
\[ \text{int} \rightarrow \text{bool}/c \]
\[ \text{int}/b \]

type b -> c with solution applied:

<table>
<thead>
<tr>
<th>Blue</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td>b -&gt; (int -&gt; bool)</td>
<td>int -&gt; (int -&gt; bool)</td>
</tr>
</tbody>
</table>

There is always a **best** solution, which we can a principal solution. The best solution is (at least as) preferred as any other solution.
Examples

Example 1

-  $q = \{a=\text{int}, b=a\}$
-  principal solution $S$: 
Example 1

- $q = \{a=\text{int}, b=a\}$
- principal solution $S$:
  - $S(a) = S(b) = \text{int}$
  - $S(c) = c$ (for all $c$ other than $a, b$)
Example 2

- \( q = \{a=\text{int}, b=a, b=\text{bool}\} \)

- principal solution \( S: \)
Example 2

- $q = \{a=\text{int}, b=a, b=\text{bool}\}$
- principal solution $S$:
  - does not exist (there is no solution to $q$)
Unification: An algorithm that provides the principal solution to a set of constraints (if one exists)

- Unification systematically simplifies a set of constraints, yielding a substitution
  - Starting state of unification process: (I,q)
  - Final state of unification process: (S, { })
Unification simplifies equations step-by-step until
• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

```
type ustate = substitution * constraints
unify_step : ustate -> ustate
```
Unification simplifies equations step-by-step until
• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

```
unify_step (S, {bool=bool} U q)   =   (S, q)
unify_step (S, {int=int}      U q)   =   (S, q)
```
Unification

Unification simplifies equations step-by-step until
• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

\[
\text{type ustate} = \text{substitution} \times \text{constraints} \\
\text{unify\_step : ustate} \rightarrow \text{ustate}
\]

\[
\text{unify\_step (S, \{bool=bool\} U q)} = (S, q) \\
\text{unify\_step (S, \{int=int\} U q)} = (S, q) \\
\text{unify\_step (S, \{a=a\} U q)} = (S, q)
\]
Unification

Unification simplifies equations step-by-step until
• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

\[
\text{type ustate} = \text{substitution} \times \text{constraints}
\]

\[
\text{unify\_step} : \text{ustate} \rightarrow \text{ustate}
\]

unify\_step (S, \{A -> B = C -> D\} U q)

= (S, \{A = C, B = D\} U q)
Unification simplifies equations step-by-step until
• there are no equations left to simplify, or
• we find basic equations are inconsistent and we fail

```plaintext
type ustate = substitution * constraints
unify_step : ustate -> ustate

unify_step (S, {A -> B = C -> D} U q)
= (S, {A = C, B = D} U q)
```
unify_step (S, \{a=s\} U q) = ([s/a] o S, [s/a]q)

when a is not in FreeVars(s)
The substitution $S'$ defined to:
do $S$ then substitute $s$ for $a$

The constraints $q'$ defined to:
be like $q$ except $s$ replacing $a$

$$\text{unify\_step} (S, \{a=s\} \cup q) = ([s/a] \circ S, [s/a]q)$$

*when $a$ is not in $\text{FreeVars}(s)$*
Recall this program:

```
fun x -> x x
```

It generates the the constraints:  $a \rightarrow a = a$

What is the solution to $\{a = a \rightarrow a\}$?
Recall this program:

```
fun x -> x x
```

It generates the the constraints: \( a \rightarrow a = a \)

What is the solution to \( \{ a = a \rightarrow a \} \)?

There is none!

Notice that \( a \) does appear in FreeVars(s)

Whenever \( a \) appears in FreeVars(s) and \( s \) is not just \( a \), there is no solution to the system of constraints.
Recall this program:

\[
\text{fun } x \rightarrow x \times x
\]

It generates the constraints: \( a \rightarrow a = a \)

What is the solution to \( \{ a = a \rightarrow a \} \)?

There is none!

“\textit{when }a\textit{ is not in }\text{FreeVars}(s)\textit{” is known as the “occurs check”}
Summary: Unification Engine

(S, \{\text{bool}=\text{bool}\} \cup q) \Rightarrow (S, q)

(S, \{\text{int}=\text{int}\} \cup q) \Rightarrow (S, q)

(S, \{a=a\} \cup q) \Rightarrow (S, q)

(S, \{A\rightarrow B = C\rightarrow D\} \cup q) \Rightarrow (S, \{A = C\} \cup \{B = D\} \cup q)

(S, \{a=s\} \cup q) \Rightarrow ([s/a] \circ S, [s/a]q) \quad \text{when } a \text{ is not in } \text{FreeVars}(s)
Inventor (1960s) of algorithms now fundamental to computational logical reasoning (about software, hardware, and other things...)

"Robinson was born in Yorkshire, England in 1930 and left for the United States in 1952 with a classics degree from Cambridge University. He studied philosophy at the University of Oregon before moving to Princeton University where he received his PhD in philosophy in 1956. He then worked at Du Pont as an operations research analyst, where he learned programming and taught himself mathematics. He moved to Rice University in 1961, spending his summers as a visiting researcher at the Argonne National Laboratory's Applied Mathematics Division. He moved to Syracuse University as Distinguished Professor of Logic and Computer Science in 1967 and became professor emeritus in 1993."

--Wikipedia
Recall: unification simplifies equations step-by-step until

- there are no equations left to simplify:

\[(S, \{ \})\]

no constraints left.  
S is the final solution!
Irreducible States

Recall: unification simplifies equations step-by-step until

• there are no equations left to simplify:
  
  \[(S, \{\})\]
  no constraints left. S is the final solution!

• or we find basic equations are inconsistent:
  • \(\text{int} = \text{bool}\)
  • \(\text{s1} \rightarrow \text{s2} = \text{int}\)
  • \(\text{s1} \rightarrow \text{s2} = \text{bool}\)
  • \(a = s\) \(\text{(s contains a)}\)

(or is symmetric to one of the above)

In the latter case, the program does not type check.
TYPE INFERENCE
MORE DETAILS
Where do we introduce polymorphic values? Consider:

$$g \ (\text{fun} \ x \ -> \ 3)$$

It is tempting to do something like this:

$$\text{(fun} \ x \ -> \ 3) : \forall a. \ a \ -> \ \text{int}$$

$$g : (\forall a. \ a \ -> \ \text{int}) \ -> \ \text{int}$$

But recall the beginning of the lecture:

if we aren’t careful, we run into decidability issues
Where do we introduce polymorphic values?

In ML languages: Only when values bound in "let declarations"

```ml
let f : forall a. a -> a = fun x -> 3 in
    g f
```

No polymorphism for `fun x -> 3`!

```ml
let f : forall a. a -> a = fun x -> 3 in
    g f
```

Yes polymorphism for `f`!
Let Polymorphism

Where do we introduce polymorphic values?

let x = v

Rule:
• if v is a value (or guaranteed to evaluate to a value without effects)
  • OCaml has some rules for this
• and v has type scheme s
• and s has free variables a, b, c, ...
• and a, b, c, ... do not appear in the types of other values in the context
• then x can have type forall a, b, c. s
Let Polymorphism

Where do we introduce polymorphic values?

```ocaml
let x = v
```

Rule:
- if `v` is a value (or guaranteed to evaluate to a value without effects)
  - OCaml has some rules for this
- and `v` has type scheme `s`
- and `s` has free variables `a, b, c, ...`
- and `a, b, c, ...` do not appear in the types of other values in the context
- then `x` can have type `forall a, b, c. s`

That’s a hell of a lot more complicated than you thought, eh?
Consider this function \( f \) – a fancy identity function:

\[
\text{let } f = \text{fun } x \rightarrow \text{let } y = x \text{ in } y
\]

A sensible type for \( f \) would be:

\[
f : \text{forall } a. \ a \rightarrow a
\]
Consider this function $f$ – a fancy identity function:

$$\text{let } f = \text{fun } x \to \text{let } y = x \text{ in } y$$

A bad (unsound) type for $f$ would be:

$$f : \forall a, b. \ a \to b$$
Consider this function f – a fancy identity function:

```
let f = fun x -> let y = x in y
```

A bad (unsound) type for f would be:

```
f : forall a, b. a -> b
```

(f true) + 7

This counterexample to soundness shows that f can’t possible be given the bad type safely.
Now, consider doing type inference:

```ml
let f = fun x -> let y = x in y
```

x : a
Now, consider doing type inference:

```ml
let f = fun x -> let y = x in y
```

Suppose we generalize and allow `y : forall a.a`
Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

- Suppose we generalize and allow $y : \forall a.a$
- Then we can use $y$ as if it has any type, such as $y : b$.
Unsound Generalization Example

Now, consider doing type inference:

```ml
let f = fun x -> let y = x in y
```

then we can use `y` as if it has any type, such as `y : b`

suppose we generalize and allow `y : forall a.a`

but now we have inferred that `(fun x -> ...) : a -> b`
and if we generalize again,
`f : forall a,b. a -> b`

That’s the bad type!
Now, consider doing type inference:

```
let f = fun x -> let y = x in y
```

suppose we generalize and allow \( y : \text{forall } a.a \)

this was the bad step – \( y \) can’t really have any type at all. Its type has got to be the same as whatever the argument \( x \) is.

\( x \) was in the context when we tried to generalize \( y \)!
The Value Restriction

let x = v

this has got to be a value to enable polymorphic generalization
Unsound Generalization Again

let x = ref [] in

x : forall a . a list ref

not a value!
Unsound Generalization Again

let x = ref [] in
x := [true];

x : forall a . a list ref

use x at type bool as if x : bool list ref

not a value!
Unsound Generalization Again

```plaintext
let x = ref [] in
x := [true];
List.hd (!x) + 3
```

- `x : forall a . a list ref`
- `use x at type bool as if x : bool list ref`
- `use x at type int as if x : int list ref`

and we crash ....
What does OCaml do?

```
let x = ref [] in
```

`x : '_weak1 list ref`

A “weak” type variable can’t be generalized.

This means “I don’t know what type this is but it can only be one particular type.”

Look for the “_” to begin a type variable name.
What does OCaml do?

```ocaml
let x = ref [] in
x := [true];
```

The "weak" type variable is now fixed as a bool and can't be anything else.

bool was substituted for '_weak during type inference.
What does OCaml do?

```ocaml
let x = ref [] in
x := [true];
List.hd (!x) + 3
```

```
x : '_weak1 list ref
x : bool list ref
Error: This expression has type bool but an expression was expected of type int
```

type error ...
One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

let \( x = \text{fun () -> ref [] in} \)

\( x : \forall 'a. \text{unit \to 'a list ref} \)

now generalization is allowed
One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

```
let x = fun () -> ref [] in
x () := [true];
```

```
x : forall 'a. unit -> 'a list ref
x () : bool list ref
```

now generalization is allowed
One other example

notice that the RHS is now a value
– it happens to be a function value
but any sort of value will do

```
let x = fun () -> ref [] in
x () := [true];
List.hd (!x ()) + 3
```

what is the result of this program?

tax now generalization

is allowed

```
x : forall 'a. unit -> 'a list ref
x () : bool list ref
x () : int list ref
```
One other example

notice that the RHS is now a value – it happens to be a function value but any sort of value will do

let \( x \) = fun () -> ref [] in

\[ x () \colon \text{bool list ref} \]

\[ \text{List.hd(!x()) + 3} \]

now generalization is allowed

\[ x \colon \forall \alpha. \text{unit} \rightarrow \alpha \text{ list ref} \]

what is the result of this program?

List.hd raises an exception because it is applied to the empty list. why?
let \( x = \text{fun} () \to \text{ref} [] \) in
\[
x () := [true];
\]
\[
\text{List.hd} (\!x ()) + 3
\]

what is the result of this program?

List.hd raises an exception because it is applied to the empty list. Why?
TYPE INFERENCE:
THINGS TO REMEMBER
Type Inference: Things to remember

Declarative algorithm: Given a context $G$, and untyped term $u$:

- Find $e$, $t$, $q$ such that $G \vdash u \Rightarrow e : t$, $q$
  - understand the constraints that need to be generated

- Find substitution $S$ that acts as a solution to $q$ via unification
  - if no solution exists, there is no reconstruction

- Apply $S$ to $e$, ie our solution is $S(e)$
  - $S(e)$ contains schematic type variables $a,b,c$, etc that may be instantiated with any type

- Since $S$ is principal, $S(e)$ characterizes all reconstructions.

- If desired, use the type checking algorithm to validate
In order to introduce polymorphic quantifiers, remember:

– Quantifiers must be on the outside only
  • this is called “prenex” quantification
  • otherwise, type inference may become undecidable

– Quantifiers can only be introduced at let bindings:
  • let x = v
  • only the type variables that do not appear in the environment may be generalized

– The expression on the right-hand side must be a value
  • no references or exceptions
Didier Rémy discovered the type generalization algorithm based on levels when working on his Ph.D. on type inference of records and variants. He prototyped his record inference in the original Caml (long before OCaml). He had to recompile Caml frequently, which took a long time. The type inference of Caml was the bottleneck: “The heart of the compiler code were two mutually recursive functions for compiling expressions and patterns, a few hundred lines of code together, but taking around 20 minutes to type check! This file alone was taking an abnormal proportion of the bootstrap cycle.”

Type inference in Caml was slow for several reasons. Instantiation of a type schema would create a new copy of the entire type -- even of the parts without quantified variables, which can be shared instead. Doing the occurs check on every unification of a free type variable (as in our eager toy algorithm), and scanning the whole type environment on each generalization increased the time complexity of inference.

“I implemented unification on graphs in O(n log n)---doing path compression and postponing the occurs-check; I kept the sharing introduced in types all the way down without breaking it during generalization/instantiation; and I introduced the rank-based type generalization.”

This efficient type inference algorithm was described in Rémy's PhD dissertation (in French) and in the 1992 technical report.