Type Inference

COS 326

David Walker

Princeton University

slides copyright 2018 David Walker

permission granted to reuse these slides for non-commercial educational purposes

Last Time: ML Polymorphism

The type for map looks like this:

```
map : (‘a -> ‘b) -> ‘a list -> ‘b list
```

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

```
map : forall ‘a, ‘b. (‘a -> ‘b) -> ‘a list -> ‘b list
```

To use a value with type forall ‘a,’b,’c . t, we first substitute types for parameters ‘a, ‘b, c’. eg:

```
map  (fun x -> x + 1) [2;3;4]
```

here, we substitute [int/’a][int/’b]
in map’s type and then use map at type
(int -> int) -> int list -> int list

Last Time

<u>Type Checking (Simple Types)</u>

A function check : context -> exp -> type

- requires function arguments to be annotated with types
- specified using formal rules. eg, the rule for function call:

$$\frac{G \mid\!- e1 : t1 \to t2 \qquad G \mid\!- e2 : t1}{G \mid\!- e1\ e2 : t2}$$

Type Schemes

A *type scheme* contains type variables that may be filled in during type inference

s ::= *a* | int | bool | s -> s

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

fun (x:s) -> e

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    -- f's argument type must equal e
```

- This time, we'll collect up equations like:

```
(a -> b) = c
```

3) Solve the equations, generating substitutions of types for var's

Example: Inferring types for map

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

Step 1: Annotate

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

```
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 ::= {s11 = s12, ..., sn1 = sn2}

- e.g.: {b = b' list, a = (b -> 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

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:

- $G \vdash u \Rightarrow e : t, q$

constraints that must be solved

type (scheme)

context

annotated expression

unannotated expression

inputs

outputs

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:

- G ⊢ u => e : t, q

constraints that must be solved

type (scheme)

context

annotated expression

unannotated expression

inputs

outputs

in OCaml:

```
gen : ctxt -> exp ->
        ann_exp * scheme * constraints
```

Constraint Generation

Simple rules:

- G ⊢ x ==> x : s, { } (if G(x) = s)
- G ⊢ 3 ==> 3 : int, { } (same for other ints)
- G ⊢ true ==> true : bool, { }
- G ⊢ false ==> false : bool, { }

Operators

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

If statements

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}

Function Application

```
G ⊢ u1 ==> e1 : t1, q1
G ⊢ u2 ==> e2 : t2, q2            (for fresh a)
-------------------------------------------------------------
G ⊢ u1 u2==> e1 e2      :      a,     q1 U q2 U {t1 = t2 -> a}
```

Function Declaration

G, x : a ⊢ u ==> e : t, q (for fresh a)

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

Function Declaration

G, f : a -> b, x : a ⊢ u ==> e : t, q   (for fresh a,b)

G ⊢ rec f(x) = u ==> rec f (x : a) : b = e   :   a -> b, q U {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 ⊢ 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 ⊢ x ==> x : s,  { }     (if G(x) = s)
```

```
G ⊢ 3 ==> 3 : int, { }
```

```
G ⊢ u1 ==> e1 : t1, q1
G ⊢ u2 ==> e2 : t2, q2             (for fresh a)
--------------------------------------------------------------
G ⊢ u1 u2==> e1 e2      :      a,     q1 U q2 U {t1 = t2 -> a}
```

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

```
G, f : a -> b, x : a ⊢ u ==> e : t, q          (for fresh a,b)
-----------------------------------------------------------------------
G ⊢ rec f(x) = u ==> rec f (x : a) : b = e      :     a -> b, q U {t = b}
```

Solving Constraints

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)
- dom(S) = set of variables s.t. S(a) $\neq$ a

Solving Constraints

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)
- dom(S) = set of variables s.t. S(a) ≠ 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:

```
a = b -> c
c = 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:

a = b -> c
c = int -> bool

solution:

b -> (int -> bool)/a
int -> bool/c
b/b

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:

```
a = b -> c
c = int -> bool
```

solution:

```
b->(int->bool)   /  a
int->bool   / c
b  / b
```

constraints with solution applied:

```
b -> (int -> bool)      =     b -> (int -> bool)
         int -> bool    =   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.

A second solution

constraints:

```
a = b -> c
c = int -> bool
```

solution 1:

```
b->(int->bool)  /  a
int->bool  /  c
b / b
```

solution 2:

```
int->(int->bool)  /  a
int->bool  /  c
int  /  b
```

Substitutions

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

constraints:

a = b -> c
c = int -> bool

solution 1:

b->(int->bool) / a
int->bool / c
b / b

solution 2:

int->(int->bool) / a
int->bool / c
int / b

type b -> c with solution applied:

b -> (int -> bool)

type b -> c with solution applied:

int -> (int -> bool)

Substitutions

solution 1:

b->(int->bool) / a
int->bool / c
b / b

solution 2:

int->(int->bool) / a
int->bool / c
int / b

type b -> c with solution applied:

b -> (int -> bool)

type b -> c with solution applied:

int -> (int -> bool)

Solution 1 is "more general" – there is more flex.

Solution 2 is "more concrete"

We prefer solution 1.

Substitutions

solution 1:

b -> (int -> bool)/a
int -> bool/c
b/b

solution 2:

int -> (int -> bool)/a
int -> bool/c
int/b

type b -> c with solution applied:

b -> (int -> bool)

type b -> c with solution applied:

int -> (int -> bool)

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

solution 1:

b -> (int -> bool)/a
int -> bool/c
b/b

solution 2:

int -> (int -> bool)/a
int -> bool/c
int/b

type b -> c with solution applied:

b -> (int -> bool)

type b -> c with solution applied:

int -> (int -> bool)

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=int, b=a}
- principal solution S:

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=int, b=a, b=bool}
- principal solution S:

Examples

Example 2

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

Unification

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

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

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

```
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)
```

```
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

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

```
unify_step (S,    {A -> B    =    C -> D}    U    q)

   =  (S,  {A = C,   B = D}    U   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

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

```
unify_step (S,    {A -> B    =    C -> D}    U    q)

   =  (S,  {A = C,   B = D}    U   q)
```

Unification

unify_step (S, {a=s} U q) = ([s/a] o S, [s/a]q)

when a is not in FreeVars(s)

Unification

the substitution S' defined to:
do S then substitute s for a

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

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

when a is not in FreeVars(s)

Occurs Check

Recall this program:

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

It generates the the constraints: a -> a = a

What is the solution to {a = a -> a}?

Occurs Check

Recall this program:

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

It generates the the constraints: a -> a = a

What is the solution to {a = a -> 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.

Occurs Check

Recall this program:

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

It generates the the constraints: a -> a = a

What is the solution to {a = a -> a}?

There is none!

"when a is not in FreeVars(s)" is known as the *"occurs check"*

Summary: Unification Engine

(S, {bool=bool} U q) ➔ (S, q)

(S, {int=int} U q) ➔ (S, q)

(S, {a=a} U q) ➔ (S, q)

(S, {A->B = C->D} U q) ➔ (S, {A = C} U {B = D} U q)

(S, {a=s} U q) ➔ ([s/a] o S, [s/a]q) *when a is not in FreeVars(s)*

The value of a classics degree

Inventor (1960s) of algorithms now fundamental to computational logical reasoning (about software, hardware, and other things…)

John Alan Robinson
1930 – 2016
PhD Princeton 1956 (philosophy)

"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

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!

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:
  - int = bool
  - s1->s2 = int
  - s1->s2 = bool
  - a = s (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

Generalization

Where do we introduce polymorphic values? Consider:

```
g (fun x -> 3)
```

It is tempting to do something like this:

```
(fun x -> 3) : forall a. a -> int
```

```
g : (forall a. a -> int) -> int
```

But recall the beginning of the lecture:

if we aren't careful, we run into decidability issues

Generalization

Where do we introduce polymorphic values?

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

```
g (fun x -> 3)
```

No polymorphism for fun x -> 3!

```
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?

```
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?

Unsound Generalization Example

Consider this function f – a fancy identity function:

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

A sensible type for f would be:

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

Unsound Generalization Example

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
```

Unsound Generalization Example

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
```

goes wrong! but if f can have the bad type,
it all type checks. This *counterexample* to soundness shows
that f can't possible be given the bad type safely

Unsound Generalization Example

Now, consider doing type inference:

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

x : a

Unsound Generalization Example

Now, consider doing type inference:

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

x : a

suppose we generalize and allow y : forall a.a

Unsound Generalization Example

Now, consider doing type inference:

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

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

x : a

suppose we generalize and allow y : forall a.a

Unsound Generalization Example

Now, consider doing type inference:

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

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

x : a

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!

Unsound Generalization Example

Now, consider doing type inference:

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

x : a

suppose we generalize and allow y : 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

not a value!

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

x : forall a . a list ref

Unsound Generalization Again

not a value!

```
let x = ref [] in

x := [true];
```

x : forall a . a list ref

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

Unsound Generalization Again

```
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

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?

```
let x = ref [] in

x := [true];
```

x : '_weak1 list ref

x : bool list ref

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?

```
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

now generalization is allowed

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

x : forall ’a. unit -> ‘a 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

now generalization
is allowed

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

x () := [true];
```

x : forall ’a. unit -> ‘a list ref

x () : bool 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

now generalization
is allowed

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

x () := [true];

List.hd (!x ()) + 3
```

x : forall 'a. unit -> 'a list ref

x () : bool list ref

x () : int list ref

what is the result of this program?

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

now generalization is allowed

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

x () := [true];

List.hd (!x ()) + 3
```

x : forall 'a. unit -> 'a list ref

x () : bool list ref

x () : int list ref

what is the result of this program?

List.hd raises an exception because it is applied to the empty list. why?

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

creates a new, different reference
every time it is called

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

x () := [true];

List.hd (!x ()) + 3
```

creates one reference

creates a second totally
different reference

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:

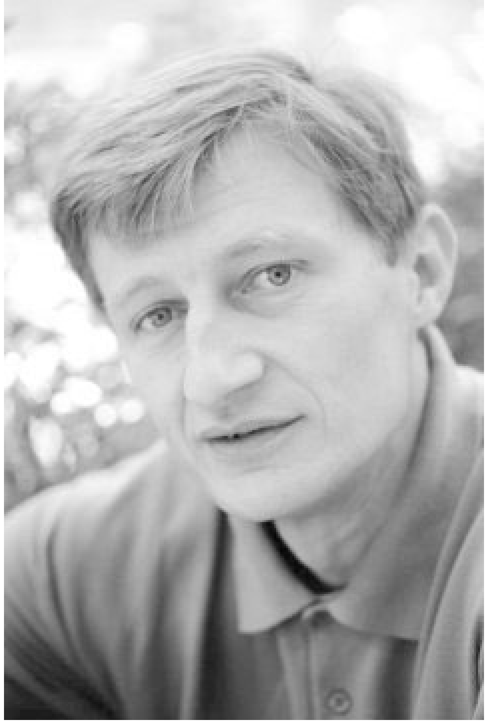
- Find e, t, q such that G ⊢ u ==> e : t, q
  - understand the constraints that need to be generated
- Find substitution S that acts as a solution to q via unification
  - if no solution exists, there is no reconstruction
- Apply S to e, ie our solution is S(e)
  - S(e) contains schematic type variables 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

Type Inference: Things to remember

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

Efficient type inference



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.

Quoted from: Oleg Kiselyov, http://okmij.org/ftp/ML/generalization.html