Type Inference

COS 326

David Walker

Princeton University

Last Time

We learned a number of facts about type inference in OCaml:

- Given an expression e, we can infer a best ("principle") type for it
- All other types for that exp are instances of the principle type
- Principle types exist because OCaml has (prenex) polymorphism
- Type inference is undecidable for general polymorphism (System F)

Last Time: Fixing My Bugs

Haskell:

- generates constraints like s1 = s2
- but also type class constraints
  - num a (“type a has number operations”)
  - eq a (“type a has = operation”)
- type inference with class constraints gives principle types
- different kind of solving engine for those constraints (not just unification)
- you don’t have to put top-level types on all Haskell definitions
  - it is just a convention
- https://www.haskell.org/ghcup/

Stephanie Weirich
Professor, U Penn

Last Time

The type inference algorithm uses type schemes, which are types with variables inside like ‘a -> ‘b

The algorithm:

- generates a type scheme for an expression
- generates constraints (scheme1 = scheme2) that must be solved for an expression to type check
- solves constraints

Last Time

The type scheme for map is:

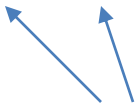
('a -> 'b) -> 'a list -> 'b list

The full polymorphic type for map is:

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

When map is *used*, we instantiate 'a and 'b with types of our choice:

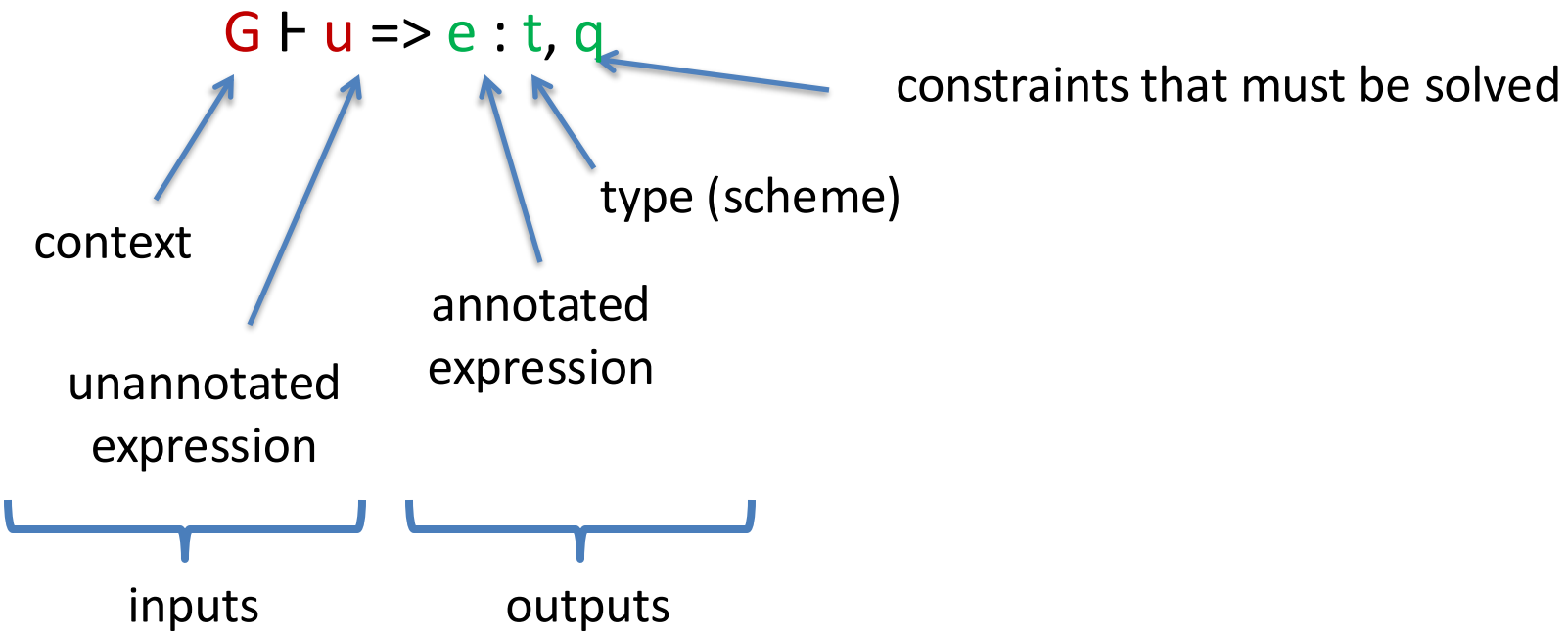
map [int,bool] : (int -> bool) -> int list -> bool list



type inference figures out which types to pick for us
may be *different* each time map is used

Last Time

We defined the type inference algorithm using a judgement with the following form:



Example rules from the inference algorithm

$$\frac{G, x : a \vdash u \Longrightarrow e : t, q \qquad \text{(for fresh } a\text{)}}{G \vdash \text{fun } x \to u \Longrightarrow \text{fun } (x : a) \to e \;:\; a \to t,\; q}$$

$$G \vdash x \Longrightarrow x : s, \{\} \qquad \text{(if } G(x) = s\text{)}$$

$$G \vdash 3 \Longrightarrow 3 : \text{int}, \{\}$$

$$\frac{G \vdash u1 \Longrightarrow e1 : t1, q1 \qquad G \vdash u2 \Longrightarrow e2 : t2, q2 \qquad \text{(for fresh } a\text{)}}{G \vdash u1\ u2 \Longrightarrow e1\ e2 \;:\; a,\; q1 \cup q2 \cup \{t1 = t2 \to a\}}$$

SOLVING CONSTRAINTS

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

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 apply a substitution S to a type scheme s.

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

to: b -> a -> b

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

Solutions

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

Solutions

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
```

Solutions

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
```

Solutions

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
```

Solutions

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
```

type b -> c with solution applied:

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

solution 2:

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

type b -> c with solution applied:

```
int -> (int -> bool)
```

Solutions

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.

Solutions

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))

Solutions

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)
```

If a solution exists, there is always a *best* solution, i.e., 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 simplifies a set of constraints
  - it looks to specialize the type variables involved, making them more concrete, generating a subsitution
  - it also looks for contradictions like “int = bool” or “a -> b = float”
    - evidence that the program can’t type check
    - unification fails

- Unification can be viewed as a computational process
  - Starting state of unification process: (Id, q)
    - Identity substitution + constraints q from type checking
  - Final state of unification process: (S, { })
  - If we find an “obviously unsolvable” equation along the way, such as “int = bool,” then we fail

Implementing Unification

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

Implementing Unification

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

Easy Cases:

- Discard equations between equal base types
- There are no contradictions here and nothing is learned

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

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

Implementing Unification

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

Easy Cases:

- Discard equations between equal type variables
- There are no contradictions here and nothing is learned

```
unify_step (S, {a=a} U q)   =   (S, q)
```

Unification

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

Recursive Cases:

- Check the top type constructor (eg: ->) is the same on each side
- Create new equations relating subparts of each type

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

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

Unification

```
type ustate = substitution * constraints

unify_step : ustate -> ustate
```

Tricky Case: equation a = s

- A variable a is equal to some other scheme s
- Idea: Eliminate a by replacing it with something equal to it -- s
- ie: substitute s for a

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

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

when a is not in FreeVars(s)

Occurs Check

Recall this program:

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

If we assume x : a for some a,

we generate the constraints: a -> a = a

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

Occurs Check

Recall this program:

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

If we assume x : a for some a,

we generate the constraints: a -> a = a

What is the solution to {a = a -> a}? There is none!

Notice that a appears 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
```

If we assume x : a for some a,

we generate the constraints: a -> a = a

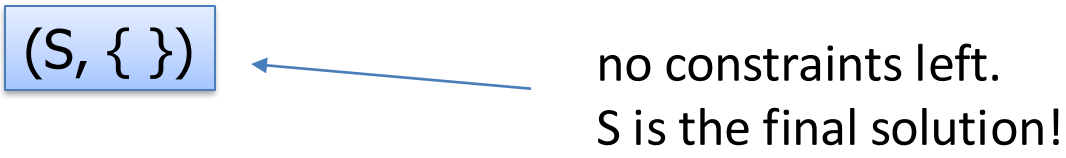
What is the solution to {a = a -> a}? There is none!

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

Irreducible States

Unification simplifies equations step-by-step until

- there are no equations left to simplify:



Irreducible States

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)

Inconsistent equations imply the program does not type 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)*

and if you get stuck before eliminating all constraints,
the program does not type check

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

ML IS CAREFULLY DESIGNED TO SUPPORT TYPE INFERENCE!
PART 3: WHICH FUNCTIONS ARE ALLOWED TO BE POLYMORPHIC?

Type Inference So Far

So far, we have inferred type schemes like:

‘a -> ‘b -> ‘b

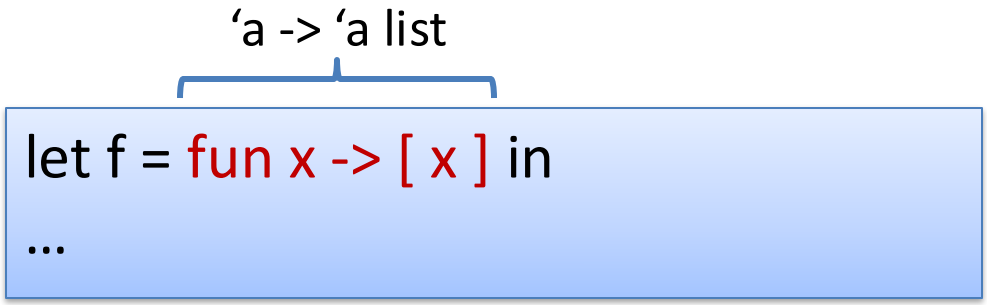
When do we convert them into full fledged polymorphic types like the following (so they can be used multiple times at different types ‘a and ‘b?

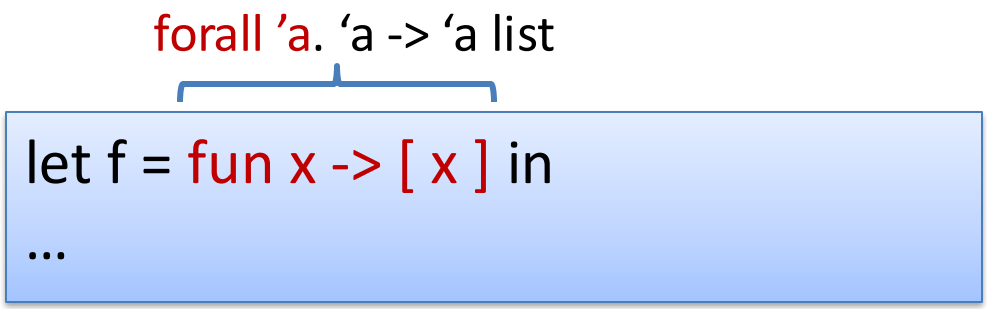
forall ‘a,b. ‘a -> ‘b -> ‘b

Generalization

Generalization converts a type scheme t with variables a, b, c into a polymorphic type forall a,c.t that quantifiers over *some subset* of them. (Which subset?)

For example





When do we introduce polymorphic values?

Where do we introduce polymorphic values? Consider:

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

It is tempting to infer a polymorphic type like this:

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

And give g a type like this:

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

But then we are inferring System F types and 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
fun x -> 3 has non-poly type
like int -> int or ‘a -> int

f : forall a. a -> int

```
let f = fun x -> 3 in
g f
```

Yes polymorphism for f!

quantifiers instantiated when f is used!

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:

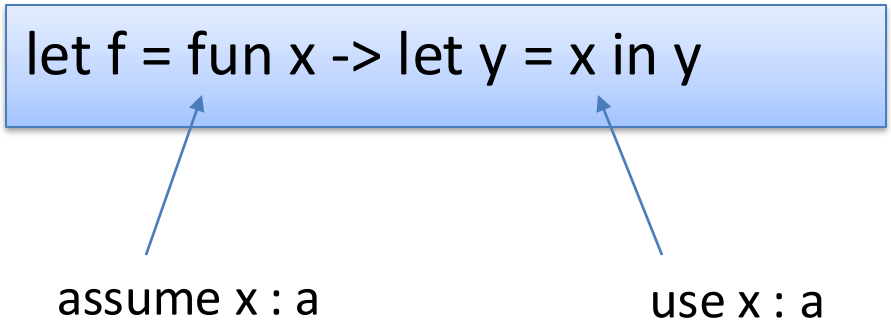
```
f : forall a, b. a -> b
```

```
(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

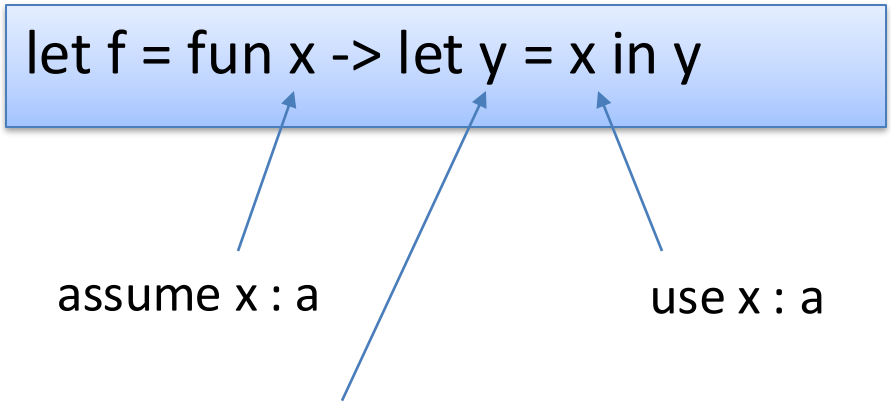
Unsound Generalization Example

Now, consider doing type inference:



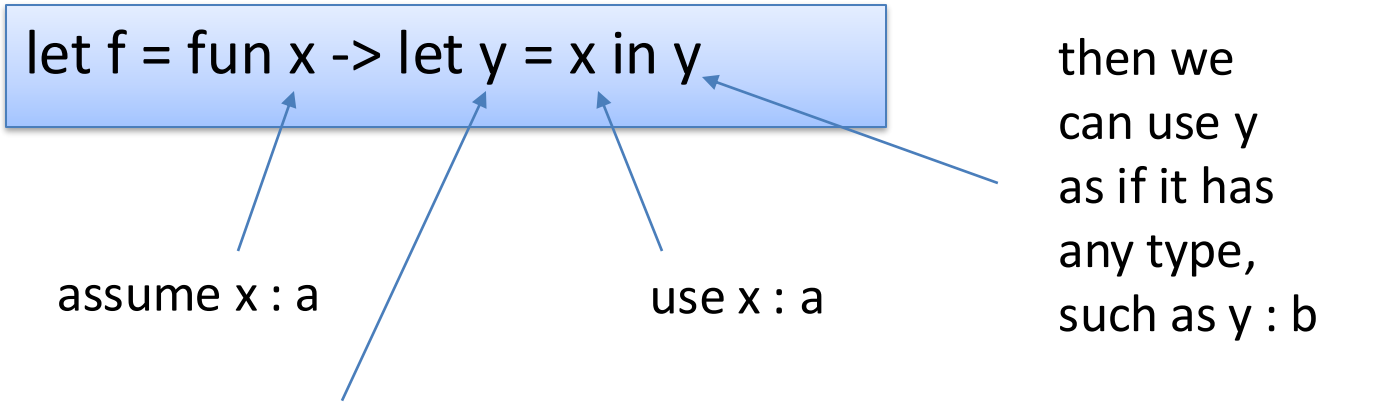
Unsound Generalization Example

Now, consider doing type inference:



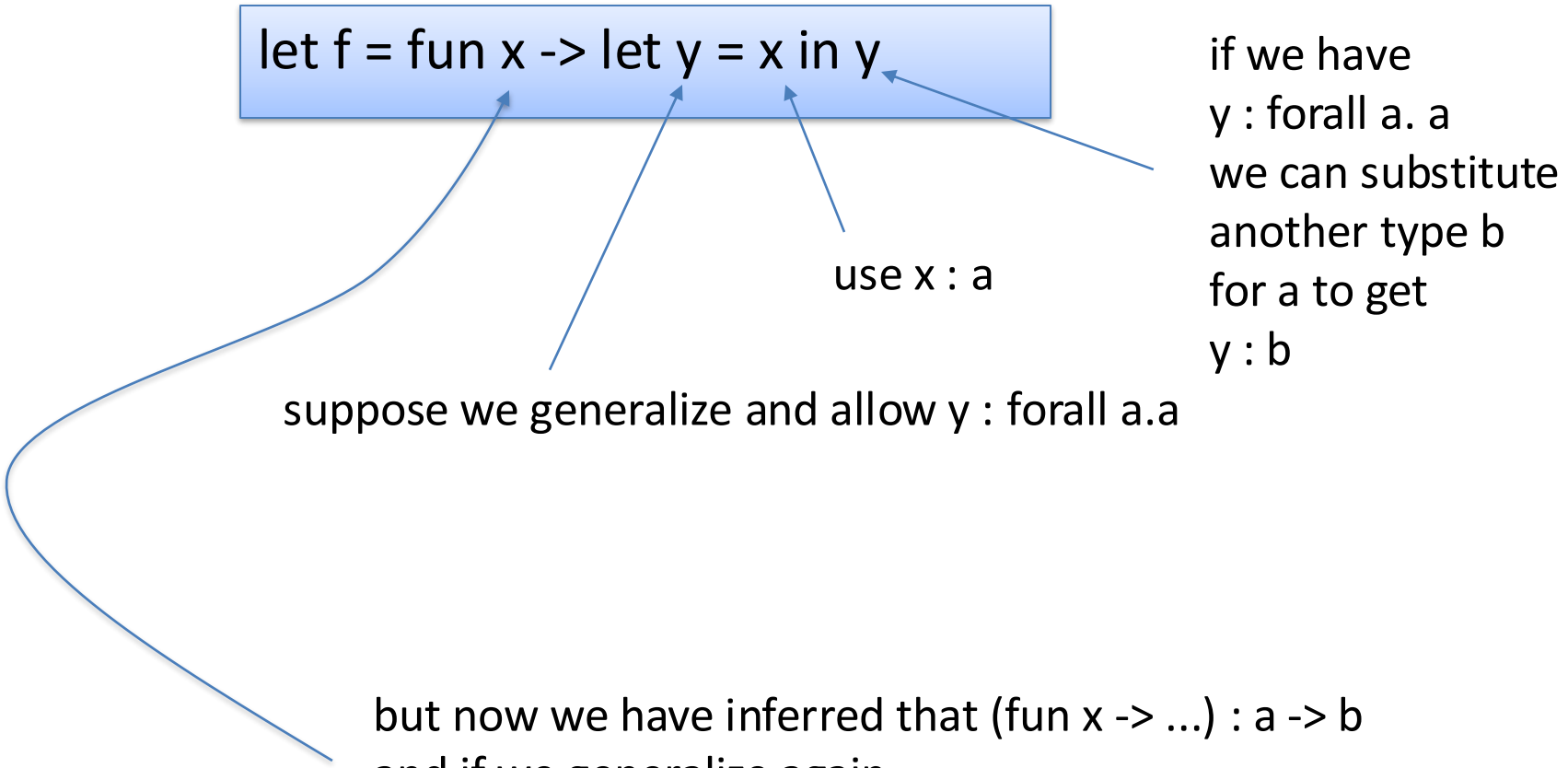
Unsound Generalization Example

Now, consider doing type inference:



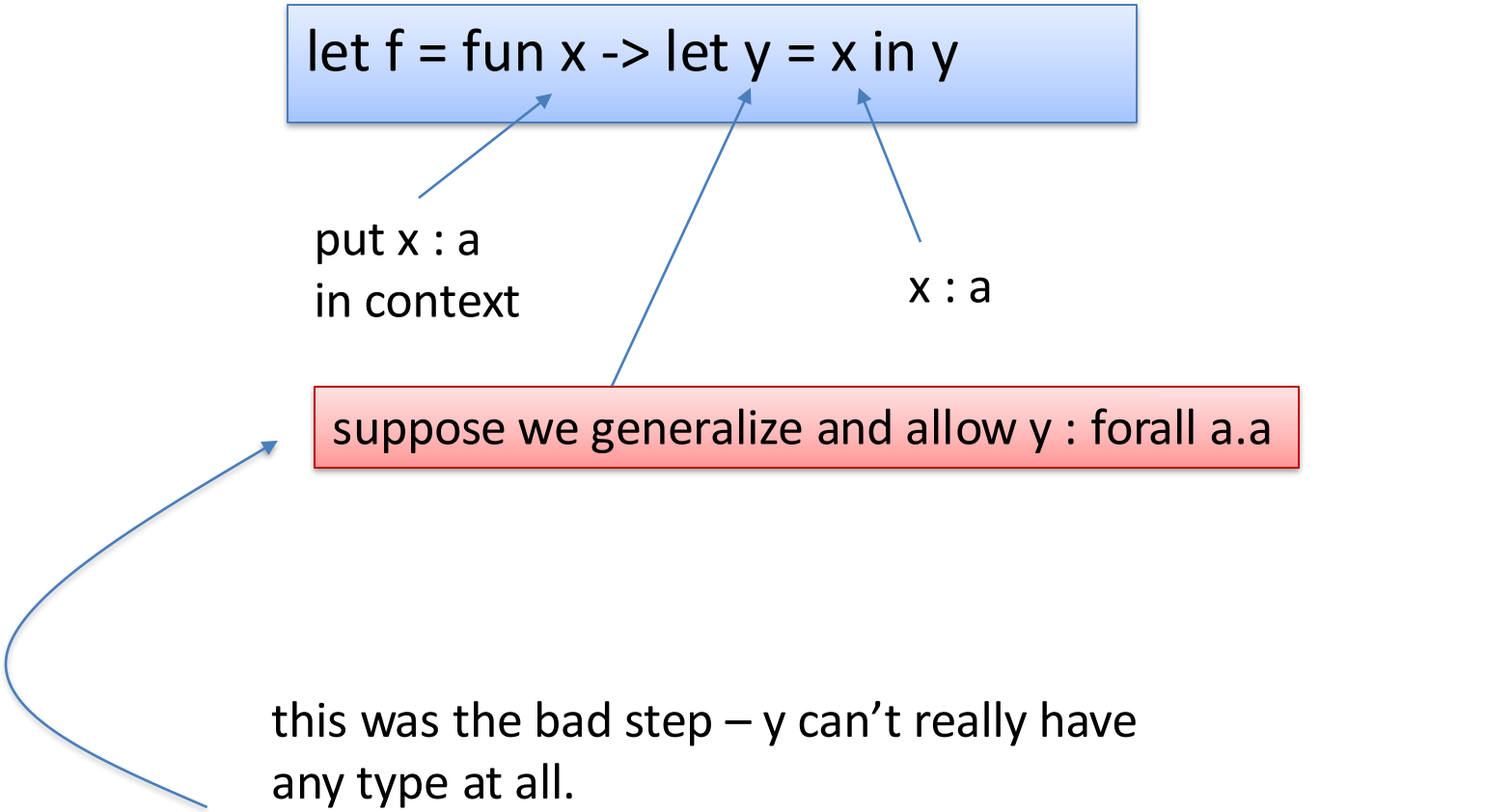
Unsound Generalization Example

Now, consider doing type inference:



Unsound Generalization Example

The bad step

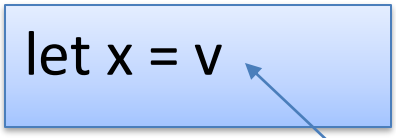


this was the bad step – y can't really have any type at all.

x : 'a was in the context so we are not allowed to generalize over 'a to get y : forall a.a

ML IS CAREFULLY DESIGNED TO SUPPORT TYPE INFERENCE!
PART 4: THE VALUE RESTRICTION

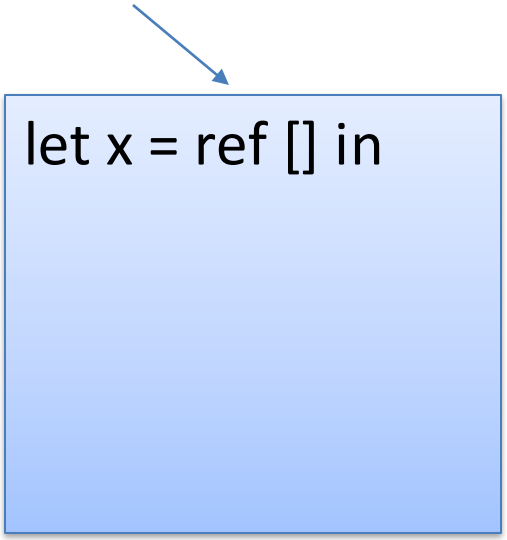
The Value Restriction



this has got to be an effect-free computation -- a *value*
to enable polymorphic generalization

Unsound Generalization Again

not a value!



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

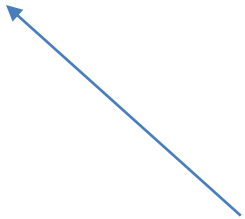
use x at type int as if x : int list ref

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

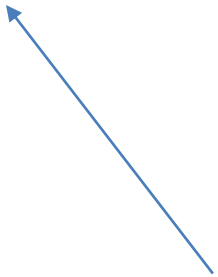
What does OCaml do?

```
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

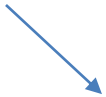
x : bool list ref

Error: This expression has type bool
but an expression was expected
of type int

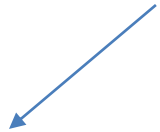
type error ...

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

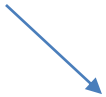


```
let x = fun () -> ref [] in
```

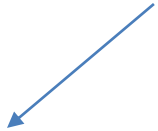
x : forall ’a. unit -> ‘a list ref

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 r1
and assigns [true]

creates a second totally
different reference r2
holding []

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, the program does not type check
- 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

Type Inference: Things to Remember

Where do we introduce polymorphic values?

```
let x = v
```

Full 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 subset of them z1,z2, z3 ... do not appear in the types in the context
- then x can have type forall z1,z2,z3. s

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