Type Checking

Part 4: Type Inference (Quantifiers)

Speaker: David Walker

COS 326

Princeton University

Last Time: A Declarative Type Inference Algorithm

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

2) Generate equations

3) Solve the equations using unification, producing a substitution for type variables, or recognize an inconsistency

... but this was an algorithm for inferring simple types: we didn't explain how or when polymorphic quantifiers could be introduced.

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 discussion from last time:

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

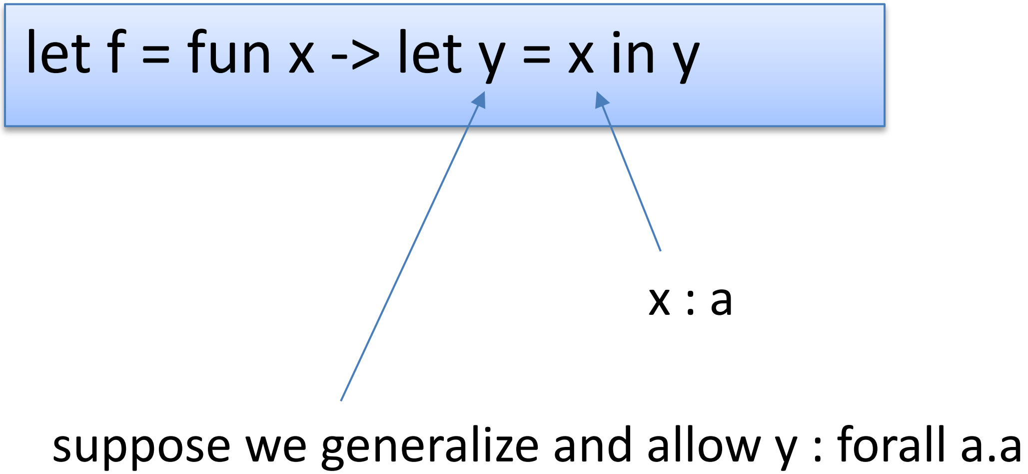
Now, consider doing type inference:

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

x : a

Unsound Generalization Example

Now, consider doing type inference:



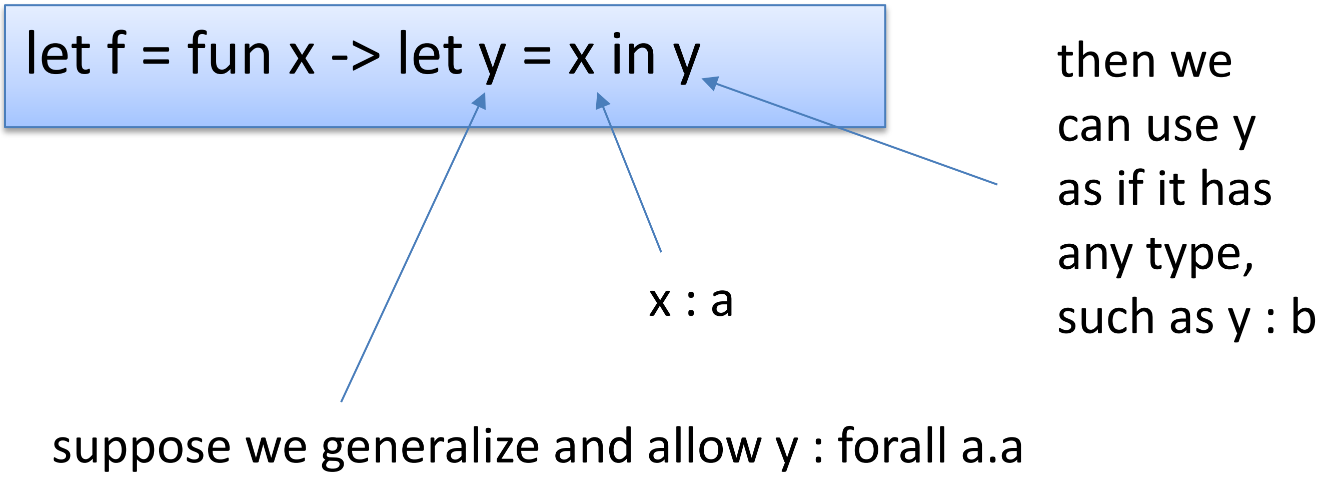
```
let f = fun x -> let y = x in y
```

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
```

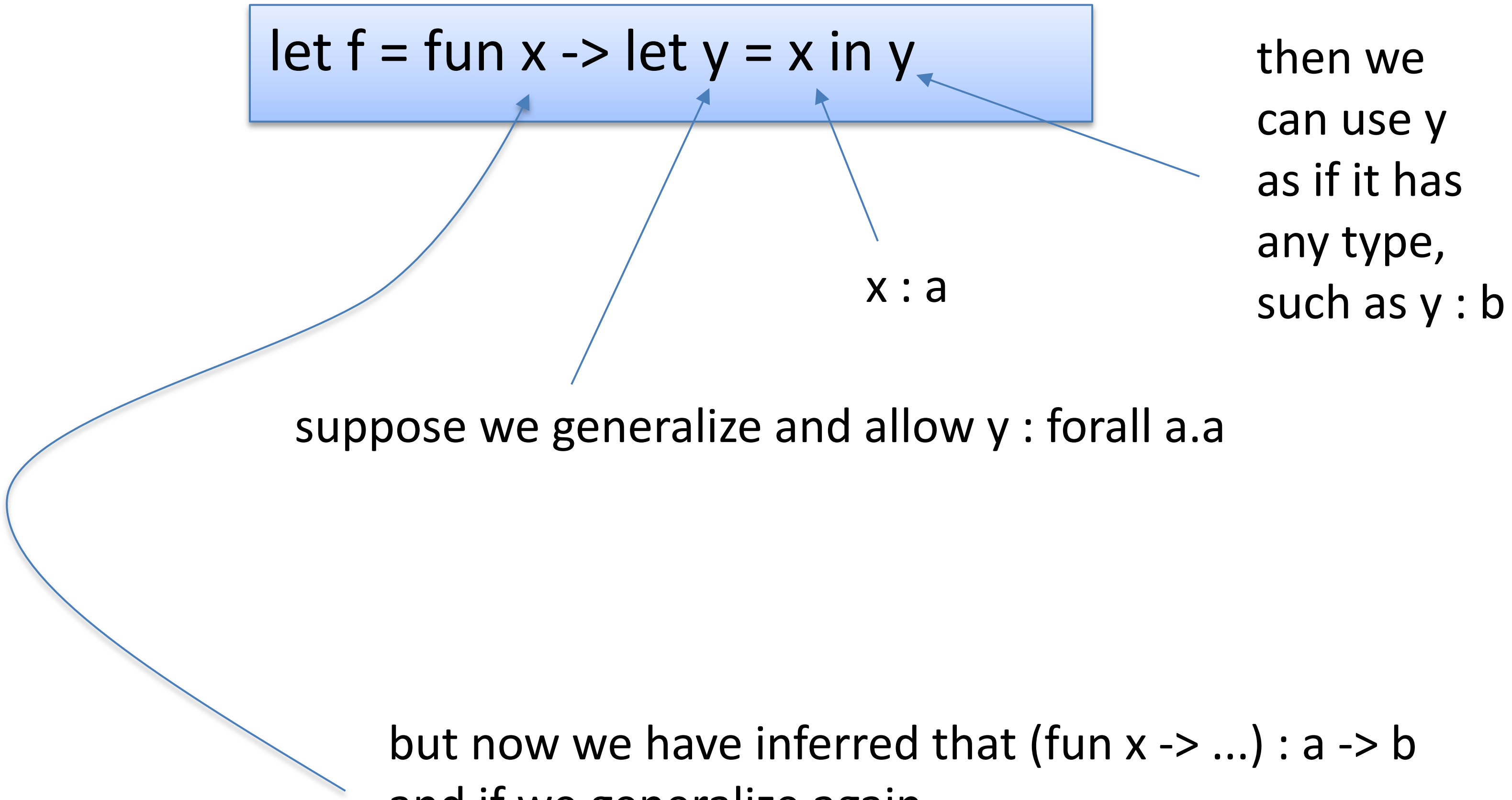
x : a

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

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
```

x : a

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

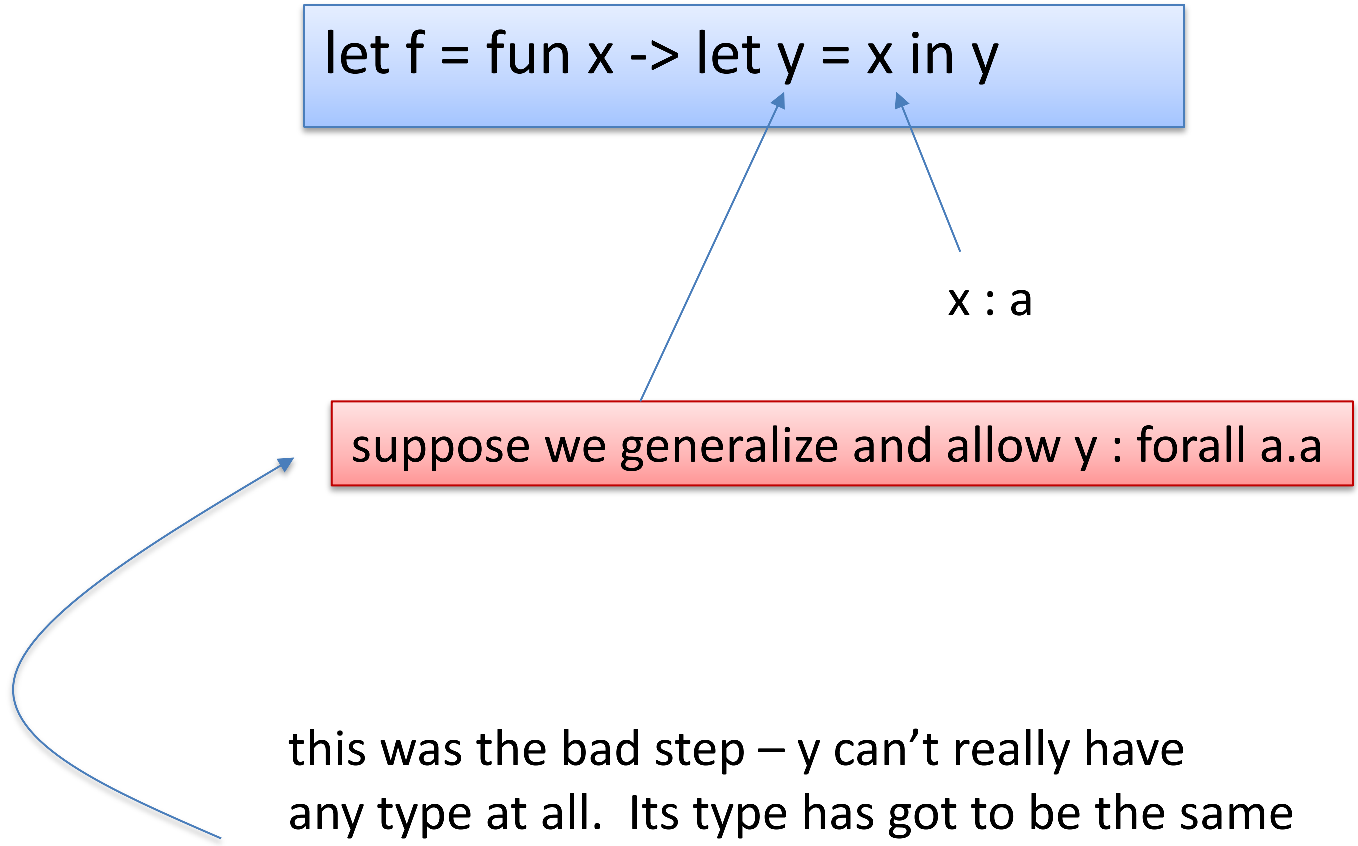
suppose we generalize and allow y : forall a.a

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!

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 nam

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 () := [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
  - in OCaml, you'll get "weak" type variables otherwise

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