Type Systems II

COS 326
David Walker
Princeton University

TYPE INFERENCE

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:

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

Last Time

Type Inference (Simple Types)

A function infer : context -> exp -> ann_exp * type * constraints

- Generates constraints (equations between types)
- Solves those constraints to find a solution (ie: a substitution)
- An example rule:

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

Last Time

Type Inference (Simple Types)

A function infer : context -> exp -> ann_exp * type * constraints

- Generates constraints (equations between types)
- Solves those constraints to find a solution (ie: a substitution)
- An example rule:

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

Up Next: How to find solutions to sets of type equations.

SOLVING CONSTRAINTS

Solving Constraints

A solution to a system of type constraints is a *substitution S*

- a function from type variables to types

Given a set of constraints:

t1 = t2   t3 = t4   t5 = t6   ...

S is a solution to these constraints when it makes LHS and RHS of each equation equal. ie:

S(t1) and S(t2) must be identical

S(t3) and S(t4) must be identical

S(t5) and S(t6) must be identical

...

Example Constraints & Solution

constraints:

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

Example Constraints & Solution

constraints:

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

solution S:

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

Example Constraints & Solution

constraints:

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

solution S:

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

Why is this a solution?

S(a) = S(b -> c) = b -> (int -> bool)

S(c) = S(int -> bool) = int -> bool

Example Constraints & Solution

constraints:

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

solution S:

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

solution S2:

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

We say that S is a more general solution than S2
because for all type t, S2(t) = U (S (t))
when U is the substitution [int/b]

Why do we like more general solutions?

constraints:

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

solution S:

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

solution S2:

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

Consider this program, which might have generated the above constraints:

```
let f : a =
  fun (x:b) : c ->
    fun n -> n < 10)
```

<u>Fact 1:</u> *Any solution* to the constraints gives rise to a *sound* type for f.

- ie: f won't crash if we give it any type that arises from a solution

<u>Fact 2:</u> If solution S is more general than S2 then f can be used in at least as many contexts (without the program crashing) if f has type S(a) than if f has type S2(a).

Why do we like more general solutions?

constraints:

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

solution S:

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

solution S2:

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

Consider this program, which might have generated the above constraints:

```
let f : a =
  fun (x:b) : c ->
    fun n -> n < 10)
```

<u>Fact 2:</u> If solution S is more general than S2 then f can be used in at least as many contexts (without the program crashing) if f has type S(a) than if f has type S2(a).

eg: with S, “f true” will type check but with S2, it won’t

Substitutions

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)
```

It turns out, there is always a *best* solution, which we can a *principle solution*. This is a pretty fortunate property – it means we can prove a kind of "completeness" property for ML type inference.

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

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 from assignment #1:

```
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 from assignment #1:

```
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 from assignment #1:

```
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"*

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 last lecture: OCaml doesn't have those sorts of types.

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 -> int = fun x -> 3 in
(f 7, f true
```

Yes polymorphism for f!

Generalization

```
let f : forall a. a -> int = fun x -> 3 in
(f 7, f true)
```

Yes polymorphism for f!

How do we use polymorphic values with type forall a.a -> int?

Each time we use them, during inference generate a fresh type variable b and use f with this type: b -> int

Because we pick a fresh variable (b, c, d, e, ...) each time, those variables can be constrained separately and take on separate types.

eg, in the first case int and in the second case bool

Using a polymorphic value by substituting a type t for a is called *type instantiation*.

Generalization: More rules!

Where do we introduce polymorphic values?

```
let x = v
```

General 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
```

General rule:

- if v is a value (or guaranteed to evaluate to a value without effects)
  - OCaml has some rules for "guaranteed to evaluate to a value"
- 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
```

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. It's 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?

And yet another example

```
let f = g x
```

Can we give f a (strong) polymorphic type?

I don't see any references around ...

And yet another example

```
let f = g x
```

Can we give f a (strong) polymorphic type?

I don’t see any references around ...

No – g could contain references. “g x” is not a value.
f will have a weakly polymorphic type (at best) in OCaml.

Watch for this in your assignment.

And yet another example

Sometimes, you can change this:

```
let f = g x
```

to something like:

this:

```
let f = fun () -> g x
```

or this:

```
let f () = g x
```

Now the right-hand side is a value (a function value)

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 way to type check the expression
  - unification will find the best (ie, the principle) solution if one exists
- Apply S to e, ie our solution is S(e)
  - S(e) contains schematic type variables a,b,c, etc
- 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
  - if x has type forall a.t, when x is used, generate fresh variable b and assume x has type t[b/a], continue type inference.
- The expression on the right-hand side must be a value
  - no references or exceptions or function calls that might contain such things

TYPE SYSTEMS:
ONE MORE THING THAT IS REALLY NIFTY

Type Checking Rules

$$\frac{}{x1{:}t1 \ \ldots \ xn{:}tn \ \vdash xi : ti}$$

"use an assumption from the context"

$$\frac{G,\ x{:}t1 \vdash e : t2}{G \vdash \lambda x{:}t.e : t1 \to t2}$$

"a function has type t1 -> t2
if when you assume x:t1, you
can show the body has type t2"

$$\frac{G \vdash e1 : t1 \to t2 \qquad G \vdash e2 : t1}{G \vdash e1\ e2 : t2}$$

"show a call has type t2
by proving the function has
type t1 -> t2 and the argument
has type t1"

Remarkably, these type checking rules are also the rules of basic (constructive) logic

Instead of thinking of “A -> B” as a function type
think of it as the logical formula “A implies B”

Logical Rules

$$\frac{}{\text{x1:t1 ... xn:tn |- xi : ti}}$$

"use an assumption from the context"

$$\frac{\text{G, x:t1 |- e : t2}}{\text{G |- }\lambda\text{x:t.e : t1 -> t2}}$$

"prove t1 -> t2 by assuming t1, and proving t2"

$$\frac{\text{G |- e1 : t1 -> t2} \qquad \text{G |- e2 : t1}}{\text{G |- e1 e2 : t2}}$$

"prove t2 by proving t1 -> t2 and by proving t1"

"modus ponens"

Logical Rules

$$\frac{}{x1{:}t1 \ldots xn{:}tn \ \vdash \ xi : ti}$$

"use an assumption from the context"

$$\frac{G,\ x{:}t1 \ \vdash \ e : t2}{G \ \vdash \ \lambda x{:}t.e : t1 \to t2}$$

"prove t1 -> t2 by assuming t1, and proving t2"

$$\frac{G \ \vdash \ e1 : t1 \to t2 \qquad G \ \vdash \ e2 : t1}{G \ \vdash \ e1\ e2 : t2}$$

"prove t2 by proving t1 -> t2 and by proving t1"

When presenting rules of logic, it is common to leave out the expressions.

Logical Proofs

Rules:

$$\frac{}{A1, \ldots, An \vdash Ai}$$

$$\frac{G, A \vdash B}{G \vdash A \to B} \qquad \frac{G \vdash A \to B \qquad G \vdash A}{G \vdash B}$$

A Proof:

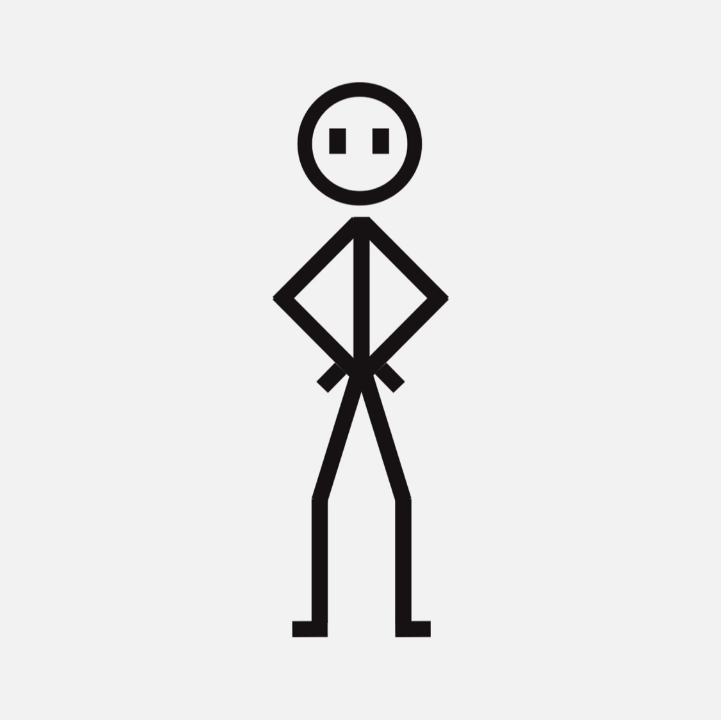
$$\cfrac{\cfrac{\cfrac{\cfrac{}{A, A \to B \vdash A \to B} \qquad \cfrac{}{A, A \to B \vdash A}}{A, A \to B \vdash B}}{A \vdash (A \to B) \to B}}{\vdash A \to (A \to B) \to B}$$

The Corresponding Program:

$$\lambda x{:}A.\ \lambda f{:}A \to B.\ f\ x$$

Curry-Howard Isomorphism

Haskell Curry



William Alvin Howard

The Curry-Howard Isomorphism is the observation that proofs and programs have similar structure.

Curry-Howard Isomorphism

Concept in Programming Languages	Concept in Logic
program	proof
type	theorem
inhabited type	true theorem
function type	implication
pair type	conjunction
union type (ie: data type)	disjunction
universal polymorphism	universal quantifier
program execution	proof simplification

Final Thoughts

There is much more to the Curry-Howard isomorphism.

- http://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf

The Curry-Howard isomorphism suggests ideas developed in logic may be useful in understanding programming languages and vice versa.

Many theorem proving/verification environments are based on the interplay between logic and programming.

Logicians were developing programming language concepts before computers existed!