CS 314: Principles of Programming Languages

Type Inference

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: Type Checking

A function check : context -> exp -> type

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

```
let f =
  fun (x:int) -> x + 1 in
f 10
```

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

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

Main Algorithm

- Add distinct variables in all places type schemes are needed

- Generate constraint (equations between types) that must be satisfied in order for an expression to type check

- Solve the equations, generating substitutions of types for the variables.

Example: Inferring types for map

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

Step 1: Annotate

constraints

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

type schemes
on functions

Step 2: Generate Constraints

b = b′ list

constraints
b = b′ list

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

Step 2: Generate Constraints

b = b′ list

constraints
b = b′ list
a = a
b = b′ list

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

a = a   b = b′ list

Step 2: Generate Constraints

b = b′ list

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

a = b′ -> a′

a = a

b = b′ list

constraints
b = b′ list
a = a
b = b′ list
a = b′ -> a′

Step 2: Generate Constraints

b = b′ list

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

a = b′ -> a′

a = a

b = b′ list

c = c′ list
a′ = c′

constraints
b = b′ list
a = a
b = b′ list
a = b′ -> a′
c = c′ list
a′ = c′

Step 2: Generate Constraints

b = b′ list

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

constraints
b = b′ list
a = a
b = b′ list
a = b′ -> a′
c = c′ list
a′ = c′
d list = c′ list

a = b′ -> a′

a = a

b = b′ list

d list = c′ list

c = c′ list
a′ = c′

Step 2: Generate Constraints

b = b′ list

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

constraints
b = b′ list
a = a
b = b′ list
a = b′ -> a′
c = c′ list
a′ = c′
d list = c′ list
d list = c
c′ list = c

a = b′ -> a′

a = a

b = b′ list

d list = c′ list

c = c′ list
a′ = c′

d list = c
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
```

constraints

b = b′ list
a = a
b = b′ list
a = b′ -> a′
c = c′ list
a′ = c′
d list = c′ list
d list = c
c′ list = c

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
```

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

- Algorithm defined as set of inference rules:
  - G ⊢ u => e : t, q

constraints that must be solved

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 ⊢ n ==> n : int, { }
  - 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 Expressions

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

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

Example

b = b′ list

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

a = b′ -> a′

Function Definition

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

Example

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

type schemes on functions

Step 2: Generate Constraints

b = b′ list

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

d list = c′ list

d list = c
c′ list = c

Function Definition

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 ⊢ n ==> n : 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)

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.

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.

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.

constraints:

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

solution:

```
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
```

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)
```

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:
  - S(a) = S(b) = int
  - S(c) = c   (for all c other than a,b)

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: ({ },q)
    - Final state of unification process: (S, { })

solution S	constraints q	solution S	constraints q
{ }	b = b′ list a = a b = b′ list a = b′ -> a′ c = c′ list a′ = c′ d list = c′ list d list = c c′ list = c	[b' -> c'/a] [b' list/b] [c' list/c]	{ }

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

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | { } -> S
    | {bool=bool} U q’ -> unify q’
    | {int = int} U q’ -> unify 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

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | { } -> S
    | {bool=bool} U q' -> unify q'
    | {int = int} U q' -> unify q'
    | {a = a} U q' -> unify 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

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
   match q with
   | …
   | {A -> B = C -> D} U q' ->
     unify 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

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {A -> B = C -> D} U q’ ->
      unify 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

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {a=s} U q' ->
       unify ([s/a] ∪ S) q'
```

??

Unification

Ideal solution :

int / p
int / q
int / r

`unify {} {`$(p \to p \to q \;=\; q \to r \to int)$`}`

`unify {} {`$(p = q)$, $(p = r)$, $(q = int)$`}`

`unify {`$[q/p]$`} {`$(p = r)$, $(q = int)$`}`

`unify {`$[q/p]$, $[r/p]$`} {`$(q = int)$`}`

`unify {`$[q/p]$, $[r/p]$, $[int/q]$`} {}`

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

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {a=s} U q' ->
      unify ([s/a] ∪ 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

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {a=s} U q' ->
       unify ([s/a] ∪ [s/a]S) [s/a]q'
```

Any occurrence of type variable a in types within S and q must be replaced by s

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

unify {} $\{(p \to p \to q \;=\; q \to r \to int)\}$

unify {} $\{(p = q), (p = r), (q = int)\}$

Ideal solution :

int / p
int / q
int / r

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

unify {} $\{(p \rightarrow p \rightarrow q = q \rightarrow r \rightarrow int)\}$

unify {} $\{(p = q), (p = r), (q = int)\}$

unify $\{[q/p]\}$ $\{(q = r), (q = int)\}$

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

`unify {}` $\{(p \to p \to q \;=\; q \to r \to int)\}$

`unify {}` $\{(p = q), (p = r), (q = int)\}$

`unify` $\{[q/p]\}$ $\{(q = r), (q = int)\}$

Unification

let rec unify S q = match q with

| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

unify {} $\{(p \rightarrow p \rightarrow q = q \rightarrow r \rightarrow int)\}$

unify {} $\{(p = q), (p = r), (q = int)\}$

unify $\{[q/p]\}$ $\{(q = r), (q = int)\}$

unify $\{[r/p], [r/q]\}$ $\{(r = int)\}$

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

`unify {}` $\{(p \to p \to q = q \to r \to int)\}$

`unify {}` $\{(p = q), (p = r), (q = int)\}$

`unify` $\{[q/p]\}$ $\{(q = r), (q = int)\}$

`unify` $\{[r/p], [r/q]\}$ $\{(r = int)\}$

Unification

let rec unify S q = match q with
| {a=s} U q' -> unify ([s/a] ∪ [s/a]S) [s/a]q'

Ideal solution :

int / p
int / q
int / r

unify {} $\{(p \rightarrow p \rightarrow q \;=\; q \rightarrow r \rightarrow int)\}$

unify {} $\{(p = q), (p = r), (q = int)\}$

unify $\{[q/p]\}$ $\{(q = r), (q = int)\}$

unify $\{[r/p], [r/q]\}$ $\{(r = int)\}$

unify $\{[int/p], [int/p], [int/r]\}$ {}

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

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {a=s} U q' ->
       unify ([s/a] ∪ [s/a]S) [s/a]q'
```

Occurs Check

```
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}
```

- Consider a program:
  - fun x -> x x

```
# fun x -> x x ;;
Line 1, characters 11-12:
Error: This expression has type 'a -> 'b but an expression was expected of type 'a
The type variable 'a occurs inside 'a -> 'b
```

  - fun (x:'a) -> ((x x):'b)

Occurs Check

```
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}
```

- Consider a program:
  - fun x -> x x

```
# fun x -> x x ;;
Line 1, characters 11-12:
Error: This expression has type 'a -> 'b but an expression was expected of type 'a
The type variable 'a occurs inside 'a -> 'b
```

  - fun (x:'a) -> ((x x):'b)

- It generates the constraints: 'a = 'a > 'b
- What is the solution to {'a = 'a > 'b}?

Occurs Check

- Consider a program:
  - fun x -> x x
- It generates the constraints: ’a = ’a > ’b
- What is the solution to {’a = ’a > ’b}?

- There is none!

For a constraint {a = s}, whenever a appears in TypeVars(s) and s is not just a, there is no solution to the system of constraints.

Occurs Check

- Consider a program:
  - fun x -> x x
- It generates the constraints: ’a = ’a > ’b
- What is the solution to {’a = ’a > ’b}?

- There is none!

“when a is not in TypeVars(s)” is known as the “occurs check”

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

```
unify : substitution -> constraints
-> substitution
```

```
let rec unify S q =
    match q with
    | …
    | {a=s} U q' ->
       unify ([s/a] ∪ [s/a]S) [s/a]q'
       when a is not in TypeVars(s)
```

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] U [s/a]S, [s/a]q) when a is not in TypeVars(s)