Type Inference (I)

Response to Criticisms of Typed Languages

- Types overly constrain functions & data
  - Polymorphism makes typed constructs useful in more contexts
    - universal polymorphism => code reuse
      - \x.x : ‘a → ‘a (* ’a is any type *)
      - reverse : ‘a list → ‘a list (* ’a is any type *)
    - existential polymorphism => modules & abstract data types
      - $T = \exists X\ \{a{:}\,X;\ f{:}\,X \rightarrow bool\}$
      - intT = {a: int; f: int → bool}
      - boolT = {a: bool; f: bool → bool}
- Types clutter programs and slow down programmer productivity
  - Type inference.
    - uninformative annotations may be omitted

Type Schemes

- A type scheme contains type variables that may be filled in during type inference
  - s ::= ‘a | int | bool | s1 → s2
  - ‘a is a type variable
- A term scheme is a term (a.k.a. expression) that contains type schemes rather than proper types
  - e ::= … | fun f (x:s1) : s2 = e
  - Note the above *named function* notation

UNTYPED LANGUAGE

- e::=

x	
| c	(consts: 0, 1, …, true, false)
| e_1 bop e_2	(binary operations)
| fun f (x) = e	(named function, can be recursive)
| e_1 e_2	(applications)

EXAMPLE

```
fun map (f, l) =
    if null (l) then
        nil
    else
        cons (f (hd l), map (f, tl l)))
```

EXAMPLE

```
fun map (f, l) =
    if null (l) then
        nil
    else
        cons (f (hd l), map (f, tl l)))
```

library functions
argument type is ‘a list

library function
argument type is (‘a *
‘a list)
result type is ‘a list

result type is ‘a

result type is ‘a list

Step 1: Add Type Schemes

```
fun map (f : a, l : b) : c =
    if null (l) then
        nil
    else
        cons (f (hd l), map (f, tl l)))
```

type schemes
on functions

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
  if null (l) then
    nil
  else
    cons (f (hd l), map (f, tl l)))
```

- walk over the program & keep track of the type equations t1 = t2 that must hold in order to type check the expressions according to the normal typing rules
- introduce new type variables for unknown types whenever necessary

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
    if null (l) then
        nil
    else
        cons (f (hd l), map (f, tl l)))
```

b = b' list

Step 2: Generate Constraints

constraints
b = b' list

```
fun map (f : a, l : b) : c =
    if null (l) then
        nil  : d list
    else
        cons (f (hd l), map (f, tl l)))
```

Step 2: Generate Constraints

constraints
b = b' list

```
fun map (f : a, l : b) : c =
  if null (l) then
    nil  : d list
  else
    cons (f (hd l), map (f, tl l)))
```

b = b'' list

b = b''' list

Step 2: Generate Constraints

constraints
b = b' list

```
fun map (f : a, l : b) : c =
    if null (l) then
        nil  : d list
    else
        cons (f (hd l), map (f, tl l: b''' list)))
```

b = b'' list

b = b''' list

STEP 2: GENERATE CONSTRAINTS

constraints
b = b' list
b = b'' list
b = b''' list

```
fun map (f : a, l : b) : c =
    if null (l) then
        nil  : d list
    else
        cons (f (hd l : b'')), map (f, tl l: b''' list)))
```

a = a   b = b''' list

Step 2: Generate Constraints

constraints
b = b' list
b = b'' list
b = b''' list
a = a
b = b''' list

```
fun map (f : a, l : b) : c =
    if null (l) then
        nil : d list
    else
        cons (f (hd l : b'') : a', map (f, tl l) : c))
```

a = b'' → a'

Step 2: Generate Constraints

constraints
b = b' list
b = b'' list
b = b''' list
a = a
b = b''' list
a = b'' -> a'

fun map (f : a, l : b) : c =
    if null (l) then
        nil : d list
    else
        cons (f (hd l) : a', map (f, tl l) : c)) : c' list

c = c' list
a' = c'

Step 2: Generate Constraints

fun map (f : a, l : b) : c =
    if null (l) then
        nil   : d list
    else
        cons (f (hd l), map (f, tl l))) : c' list

d list = c' list

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

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
    if null (l) then
        nil
    else
        cons (f (hd l), map (f, tl l))
    : d list
```

d list = c

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

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
    if null (l) then
        nil
    else
        cons (f (hd l), map (f, tl l)))
```

final
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

- Constraint solution provides all possible solutions to type scheme annotations on terms

final
constraints
b = b' list
b = b'' list
b = b''' list
a = a
...

solution
a = b' → c'
b = b' list
c = c' list

map (f : b' → c'
x : b' list)
: c' list
=
...

Step 4: Generate types

- Generate types from type schemes
  - Option 1: pick an instance of the most general type when we have completed type inference on the entire program
    - map : ((int → int) * int list) → int list
  - Option 2: generate polymorphic types for program parts and continue (polymorphic) type inference
    - map : ∀(a,b) ((a → b) * a list) → b list

Quiz: Generating Types

Generate the polymorphic types for the following function:

```
fun fold (f, a, l) =
   case l of
            nil => a
          | h::t => fold (f, f (h, a), t)
```

Type Inference Details

- Type constraints are sets of equations between type schemes
  - q ::= {s11 = s12, ..., sn1 = sn2}
  - eg: {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
- Algorithm defined as set of inference rules (as always).
- Judgement form:
  - G |-- u ==> e : t, q
  - u is untyped expression
  - e : t is a term scheme
  - q is a set of constraints

Constraint Generation

- Simple rules:
  - G |-- x ==> x : s, { }   (if G(x) = s)
    - If G(x) is not defined then x is free variable
  - 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: a,
                                   q1 U q2 U q3 U {t1 = bool, a = t2, a = t3}
```

Function Application

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

Function Declaration

```
G, f : a -> b, x : a |-- u ==> e : t, q
----------------------------------------------------------------
G |-- fun f(x) = u  ==> fun f (x : a) : b = e

                  : a -> b, q U {t = b}
```

(a, b are fresh type variables; not in G)

Solving Constraints

- A solution to a system of type constraints is a substitution S
  - a **function** from *type variables* to *type schemes*
  - substitutions are defined on all type variables (a total function), but only some of the variables are actually changed:
    - S(a) = a (for almost all variables a)
    - S(a) = s (for some a and some type scheme s)
  - dom(S) = set of variables s.t. S(a) $\neq$ a

Substitutions

- Given a substitution S, we can define a function S* from type schemes (as opposed to type variables) to type schemes:
  - S*(int) = int
  - S*(s1 → s2) = S*(s1) → S*(s2)
  - S*(a) = S(a)

  - For simplicity, next I will write S(s) instead of S*(s)
  - s denotes type schemes, whereas a, b, c denote type variables
  - This function replaces all type variables in a type scheme.

Composition of Substitutions

- Composition (U o S) applies the substitution S and then applies the substitution U:
  - (U o S)(a) = U(S(a))
- We will need to compare substitutions
  - T <= S if T is "more specific" than S
  - T <= S if T is "less general" than S
  - Formally: T <= S if and only if T = U o S for some U

Composition of Substitutions

- Examples:
  - example 1: any substitution is less general than the identity substitution I:
    - S <= I because S = S o I
  - example 2:
    - S(a) = int, S(b) = c → c
    - T(a) = int, T(b) = c → c, T(c) = int
    - we conclude: T <= S
    - if T(a) = int, T(b) = int → bool then T is unrelated to S (neither more nor less general)

Solving a Constraint

- Judgment format: S |= q
  (S is a solution to the constraints q)

```
----------
S |= { }
```

any substitution is
a solution for the empty
set of constraints

```
S(s1) = S(s2)        S |= q
---------------------------------
S |= {s1 = s2} U q
```

a solution to an equation
is a substitution that makes
left and right sides equal

Most General Solutions

- S is the principal (most general) solution of a set of constraints q if
  - S |= q (S is a solution)
  - if T |= q then T <= S (S is the most general one)
- Lemma: If q has a solution, then it has a most general one
- We care about principal solutions since they will give us the most general types for terms (polymorphism!)

- Exercise:

Prove: If q has a solution, then it has a most general one.

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:
    - does not exist (there is no solution to q)