Type Inference (II)

Solving Constraints (Recap)

- Judgement form:
  - G |-- u ==> e : t, q
  - u is untyped expression
  - e : t is a term scheme
  - q is a set of 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 most variables a)
    - S(a) = s (for some a and some type scheme s)
  - dom(S) = set of variables s.t. S(a) ≠ 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*(bool) = bool
  - 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.
- There's no variable binding in the language of type scheme, hence no danger of capturing!

Extensions to Substitution

- Substitution can be extended pointwise to the typing context:

  G := . | G, x : s

  S(.) = .

  S(G, x:s) = S(G), x: S(s)

Similarly, substitution can be applied to the type annotations in an expression, e.g.:

S(x) = x

S(\x:s.e) = \x:S(s).S(e)

S(nil[s]) = nil[S(s)]

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)

Preservation of typing under type substitution

- Theorem: If S is any type substitution and

  G |- e : s, then S(G) |- S(e) : S(s)

Proof: straightforward induction on the typing derivations.

Solving a Constraint (first attempt)

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

However this will not help you Solve q to obtain S!

```
----------
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!)

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)

PRINCIPAL SOLUTIONS

- principal solutions give rise to most general *reconstruction* of typing information for a term:
  - fun f(x:a):a = x
    - is a most general reconstruction
  - fun f(x:int):int = x
    - is not

Unification

- Unification: An algorithm that provides the principal solution to a set of constraints (if one exists)
  - If one exists, it will be principal

Unification

- Unification: Unification systematically simplifies a set of constraints, yielding a substitution
- During simplification, we maintain (S, q)
  - S is the solution so far
  - q are the constraints left to simplify
  - Starting state of unification process: (I, q)
  - Final state of unification process: (S, { })

identity substitution is most general

Unification Machine

- We can specify unification as a transition system:
  - (S, q) -> (S′, q′)
- Base types & simple variables:

```
-------------------------------- (u-int)
(S,{int=int} U q) -> (S, q)
```

```
------------------------------ (u-eq)
(S,{a=a} U q) -> (S, q)
```

```
------------------------------------ (u-bool)
(S,{bool=bool} U q) -> (S, q)
```

Unification Machine

- Functions:

```
--------------------------------------------  (u-fun)
(S, {s11 -> s12= s21 -> s22} U q) ->
(S, {s11 = s21, s12 = s22} U q)
```

- Variable definitions

```
------------------------------------------- (a not in FV(s)) (u-var1)
(S,{a=s} U q) -> ([a=s] o S, q[s/a])
```

```
------------------------------------------ (a not in FV(s)) (u-var2)
(S,{s=a} U q) -> ([a=s] o S, q[s/a])
```

Occurs Check

- What is the solution to {a = a → a}?
  - There is none!
  - The occurs check detects this situation

```
---------------------------------------- (a not in FV(s))
(S,{a=s} U q) -> ([a=s] o S, q[s/a])
```

occurs check

Irreducible States

- Recall: final states have the form (S, { })
- Stuck states (S,q) are such that every equation in q has the form:
  - int = bool
  - s1 → s2 = s   (s not function type)
  - a = s   (s contains a)
  - or is symmetric to one of the above
- Stuck states arise when constraints are unsolvable

Termination

- We want unification to terminate (to give us a type reconstruction algorithm)
- In other words, we want to show that there is no infinite sequence of states
  - (S1,q1) → (S2,q2) → …
- Theorem: unification algorithm always terminates.

Termination

- We associate an ordering with constraints
  - q < q' if and only if
    - q contains fewer equations than q'
    - q contains the same number of equations but fewer variables than q'
    - q contains the same number of variables as q' but fewer type constructors (ie: fewer occurrences of int, bool, or "→")
    - in other words, q is simpler than q'
  - This is a lexicographic ordering on (nq, nv, nc)
    - nq: Number of equations
    - nv: Number of variables
    - nc: Number of constructors
    - There is no infinite decreasing sequence of constraints
  - To prove termination, we must demonstrate that every step of the algorithm reduces the size of q according to this ordering

Termination

- Lemma: Every step reduces the size of q
  - Proof: By observation on the definition of the reduction relation.

```
--------------------------------
(S,{int=int} U q) -> (S, q)

----------------------------------------------
(S,{s11 -> s12= s21 -> s22} U q) ->
(S, {s11 = s21, s12 = s22} U q)

------------------------------------
(S,{bool=bool} U q) -> (S, q)

------------------------ (a not in FV(s))
(S,{a=s} U q) ->
([a=s] o S, q[s/a])

-----------------------------
(S,{a=a} U q) -> (S, q)

------------------------ (a not in FV(s))
(S,{s=a} U q) ->
([a=s] o S, q[s/a])
```

Correctness

- we know the algorithm terminates
- we want to prove that a series of steps:

(I, q1) -> (S2, q2) -> (S3, q3) -> … -> (S, {})

solves the initial constraints q1

- We'll do that by induction on the length of the unification sequence, but we'll need to define the invariants that are preserved from step to step

Complete Solutions

- A complete solution for (S, q) is a substitution T such that
  1. T <= S
  2. T |= q
  - intuition: T extends S and solves q

- A principal solution T for (S, q) is complete for (S, q) and
  3. for all T' such that 1. and 2. hold, T' <= T
  - intuition: T is the most general solution (it's the least restrictive)

Properties of Solutions

- Lemma 1: Every final state (S, { }) has a complete and principal solution, which is S. (note: "every" means regardless of the length of unification steps)
- To show that S is a complete solution:
    - S <= S
    - S |= { }

every substitution is a solution to the empty set of constraints

- To show that S is a principal solution for (S, {}):
    - For any other complete solution T:
        - T <= S
    - Therefore, S is the principal solution.
- Proof: by induction on the length of the unification sequence.
    - Case 0 steps: S |= {} is always true for any S, including I. S<= I for any S.
    - Hypothesis: for a sequence of k steps starting from (S', q), final state (S, {}) has a complete solution S, i.e. S<=S', S|=q.

- Case k+1 steps:
  - There are 6 subcases, one for each unification rule.
  - Cases int, bool, fun and equal are trivial since S' remains the same after the first step, then remaining k steps is true due to hypothesis.
  - Case (u-var1) and (u-var2):

    if ([a=s] o S, q[s/a]) has a complete solution T, i.e.,

    T<=[a=s] o S, and T |= q[s/a] (by IH);

    then (S, {s=a} U q) also has complete solution T, because

    T<=S, and since T<=[a=s] o S, T |= {a=s} U q

  (proved)

```
---------------------------------
(S,{int=int} U q) -> (S, q)

------------------------------------------------
(S,{s11 -> s12= s21 -> s22} U q) ->
(S, {s11 = s21, s12 = s22} U q)

-------------------------------------
(S,{bool=bool} U q) -> (S, q)

--------------------------------------------------- (a not in FV(s))
(S,{a=s} U q) -> ([a=s] o S, q[s/a])

------------------------------
(S,{a=a} U q) -> (S, q)

--------------------------------------------------- (a not in FV(s))
(S,{s=a} U q) -> ([a=s] o S, q[s/a])
```

Properties of Solutions

- Lemma 2: No stuck state has a complete solution (or any solution at all)
  - it is impossible for a substitution to make the necessary equations equal
    - int $\neq$ bool
    - int $\neq$ t1 -> t2
    - ...

Properties of Solutions

- Lemma 3
  - If (S, q) -> (S', q') then
    - T is complete for (S,q) iff T is complete for (S',q')
    - T is principal for (S,q) iff T is principal for (S',q')
  - Proof: by induction on the derivation of unification step ->
  - In the forward direction, this is the preservation theorem for the unification machine!

SUMMARY: UNIFICATION

- By termination, (I, q) →* (S, q') where (S, q') is irreducible. Moreover:

  If q' = { } then:
  - (S, q') is final (by definition)
  - S is a principal solution for q
    - Consider any T such that T is a solution to q.
    - Now notice, S is principal for (S, q') (by lemma 1)
    - S is principal for (I, q) (by lemma 3)
    - Since S is principal for (I, q), we know T <= S and therefore S is a principal solution for q.

SUMMARY: UNIFICATION (CONT.)

- … Moreover:
  - If q' is not { } (and (I, q) →* (S, q') where (S, q') is irreducible) then:
  - (S, q') is stuck. Consequently, (S,q') has no complete solution. By lemma 3, even (I, q) has no complete solution and therefore q has no solution at all.

Summary: Type Inference

- Type inference algorithm.
  - Given a context G, and untyped term u:
    - Find e, t, q such that G |- u ==> e : t, q
    - Find principal solution S of q via unification
      - if no solution exists, there is no reconstruction
    - Apply S to e, i.e., 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.

Let Polymorphism

- Generalized from the type inference algorithm
- A.k.a ML-style or Hindley Milner-style polymorphism
- Basis of "generic libraries":
  - Trees, lists, arrays, hashtables, streams, …
- let id = \x. x in

      (id 25, id true)
  - id can't be both int → int and bool → bool, due to:

$$\frac{G \vdash e1 : t1 \quad G, x{:}t1 \vdash e2 : t2}{G \vdash \text{let } x\text{=}e1 \text{ in } e2 : t2} \ [\text{t-let}]$$

Let Polymorphism

- Instead:

$$\frac{G \vdash e2[e1/x] : t2 \quad G \vdash e1 : t1}{G \vdash \text{let } x{=}e1 \text{ in } e2 : t2} \quad [\text{t-letPoly}]$$

- Or using the constraint generation rule:

```
G |-- u2[u1/x] ==> e2[e1/x] : t2, q2
G |-- u1 ==> e1 : t1, q1
-----------------------------------------------------------------------
G |-- let x = u1 in u2 ==> let x = e1 in e2: t2, q1 U q2
```

Caveat with Let Polymorphism

- If the body (e2) contains many let bindings
- Every occurrence of a let binding in e2 causes a type check of right-hand-side e1
- e1 itself can contain many let binding as well
- Time complexity **exponential** to the size of the expression!
- Practical implementation uses a smarter but equivalent algorithm:
  - Amortized linear time
  - Worse-case still exponential
  - see Pierce Ch. 22.