Lesson 10
Type Reconstruction

2/26

Chapter 22

Type Reconstruction

- substitutions
- typing with constraint sets (type equations)
- unification: solving constraint sets
- principlal types
- let polymorphism

Type substitutions

Language: □□ [Bool, Nat] with type variables

A type substitution □ is a finite mapping from type variables to types.

□ = [X => Nat -> Nat, Y => Bool, Z => X -> Nat]

Type substitutions can be applied to types: □T

□(X -> Z) = (Nat -> Nat) -> (X -> Nat)

This extends pointwise to contexts: □ □

Type substitutions

Language: □□ [Bool, Nat] with type variables

A type substitution □ is a finite mapping from type variables to types.

□ = [X => Nat -> Nat, Y => Bool, Z => X -> Nat]

Type substitutions can be applied to types: □T

□(X -> Z) = (Nat -> Nat) -> (X -> Nat)

This extends pointwise to contexts: □ □

Composition of substitutions

□ ∘ □(X) = □ (□X) if X □ dom □

= □ X otherwise

Substitutions and typing

Thm: If □ |- t: T, then □□ |- □t: □T for any type subst. □.

Prf: induction on type derivation for □ |- t: T.

"Solving" typing problems

Given □ and t, we can ask:

1. For every □, does there exist a T s.t. □□ |- □t: T?

2. Does there exist a □ and a T s.t. □□ |- □t: T?

Question 1 leads to polymorphism, where T = □T' and □ |- t: T'. The type variables are "quantified".

Question 2 is the basis for type reconstuction: we think of the type variables as unknowns to be solved for.

Defn: A solution for (□,t) is a pair (□, T) s.t. □□ |- □t: T.

Example: solutions of a typing problem

(∅, □x:X. □y:Y. □z:Z . (x z) (y z)) has solutions

[X => Nat -> Bool -> Nat, Y => Nat -> Bool, Z => Nat]

[X => X1 -> X2 -> X3, Y => X1 -> X2, Z => X1]

Constraints

A constraint set C is a set of equations between types.
$C = \{S_i = T_i \mid i$ □ $1,..,n\}$.

A substitution □ unifies (or satisfies) a constraint set C
if □S_i = □T_i for every equation $S_i = T_i$ in C.

A constraint typing relation □ |- t: T | C $\mathcal{X}$ where $\mathcal{X}$ is a set of "fresh" type variables used in the constraint set C. This relation (or judgement) is defined by a set of inference rules.

Constraint inference rules

Inference rule for application

$$\frac{\begin{array}{c} \square \vdash t_1 : T_1 \mid C_1\ \mathcal{X}_1 \qquad \square \vdash t_2 : T_2 \mid C_2\ \mathcal{X}_2 \\ \mathcal{X}_1 \square\ \mathcal{X}_2 = \mathcal{X}_1 \square\ FV(T_2) = \mathcal{X}_2 \square\ FV(T_1) = \varnothing \\ X \square\ \mathcal{X}_1, \mathcal{X}_2, t_1, t_2, T_1, T_2, C_1, C_2, \square \\ C = C_1 \square\ C_2 \square\ \{T_1 = T_2 \to X\} \\ \mathcal{X} = \mathcal{X}_1 \square\ \mathcal{X}_2 \square\ \{X\} \end{array}}{\square \vdash t_1\,t_2 : X \mid C\ \mathcal{X}}$$

Constraint solutions

Defn: Suppose □ |- t: S | C $\mathcal{X}$. A solution for (□, t, S, C) is a pair (□, T) s.t. □ satisfies C and T = □S.

Thm: [Soundness of Constraint Typing]
Suppose □ |- t: S | C X. If (□, T) is a solution for (□, t, S, C) then it is also a solution for (□, t), i.e. □□ |- □t: T.

Thm: [Completeness of Constraint Typing]
Suppose □ |- t: S | C $\mathcal{X}$. If (□, T) is a solution for (□, t) then there is a solution (□', T) for (□, t, S, C) s.t. □'\$\mathcal{X}$= □.

Cor: Suppose □ |- t: S | C $\mathcal{X}$. There is a soln for (□, t) iff there is a solution for (□, t, S, C).

Unification

Defn: $\sigma < \sigma'$ if $\sigma' = \gamma \circ \sigma$ for some γ.

Defn: A principle unifier (most general unifier) for a constraint set C is a substitution σ that satisfies C s.t. $\sigma < \sigma'$ for any other σ' that satifies C.

Unification algorithm

```
unify C =
    if C = ∅ then [ ]
    else let {S = T} ∪ C' = C in
        if S = T then unify(C')
        else if S = X and X ∉ FV(T)
          then unify([X => T]C') ∘ [X => T]
        else if T = X and X ∉ FV(S)
          then unify([X => S]C') ∘ [X => S]
        else if S = S1 -> S2 and T = T1 -> T2
          then unify(C' ∪ {S1 = T1, S2 = T2})
        else fail
```

Thm: unify always terminates, and either fails or returns the principal unifier if a unifier exists.

Principal Types

Defn: A principal solution for (Γ, t, S, C) is a solution (σ, T) s.t. for any other solution (σ', T') we gave $\sigma < \sigma'$.

Thm: [Principal Types]
If (Γ, t, S, C) has a solution, then it has a principal one. The unify algorithm can be used to determine whether (Γ, t, S, C) has a solution, and if so it calculates a principal one.

Implicit Annotations

We can extend the syntax to allow lambda abstractions without type annotations: $\lambda x.t$.

The corresponding type constraint rule supplies a fresh type variable as an implicit annotation.

$$\frac{X \notin \mathcal{X} \qquad \Gamma, x{:}\ X \mid\!- \ t_1 : T \mid C\ \mathcal{X}}{\Gamma, \ \mid\!- \ \lambda x{:}\ X.\ t_1 : X \to T \mid C\ (\mathcal{X} \cup \{X\})} \quad \text{(CT-AbsInf)}$$

Let Polymorphism

let double = λf: Nat -> Nat. λx: Nat. f(f x)
in double (λx: Nat. succ x) 2

let double = λf: Bool -> Bool. λx: Bool. f(f x)
in double (λx: not x) false

An attempt at a *generic* double:

let double = λf: X -> X. λx: X. f(f x)
in let a = double (λx: Nat. succ x) 2
in let b = double (λx: not x) false

==> X -> X = Nat -> Nat = Bool -> Bool

Macro-like let rule

let double = λf: X -> X. λx: X. f(f x)
in let a = double (λx: Nat. succ x) 2
in let b = double (λx: not x) false

could be typed as:

let a = (λf: X -> X. λx: X. f(f x)) (λx: Nat. succ x) 2
in let b = (λf: X' -> X'. λx: X'. f(f x)) (λx: not x) false

or, using implicit type annotations:

let a = (λf. λx. f(f x)) (λx: Nat. succ x) 2
in let b = (λf. λx. f(f x)) (λx: not x) false

Macro-like let rule

$$\frac{\Gamma \mid\!- t_1 : T_1 \qquad \Gamma \mid\!- [x \Rightarrow t_1]t_2 : T_2}{\Gamma \mid\!- \text{let } x = t_1 \text{ in } t_2 : T_2} \quad \text{(T-LetPoly)}$$

The substitution can create multiple independent copies of t_1, each of which can be typed independently (assuming implicit annotations, which introduce separate type variables for each copy).

Type schemes

Add quantified type schemes:

T ::= X | Bool | Nat | T -> T
P ::= T | $\forall X . P$

Contexts become finite mappings from term variables to type schemes:

Γ ::= $\varnothing$ | $\Gamma, x : P$

Examples of type schemes:

Nat, X -> Nat, $\forall X.\ X$ -> Nat, $\forall X. \forall Y.\ X$ -> Y -> X

let-polymorphism rules

$$\frac{\Gamma \mid\!- \; t_1 : T_1 \qquad \Gamma, x : \forall \mathcal{X}.T_1 \mid\!- \; t_2 : T_2}{\Gamma \mid\!- \; \text{let } x = t_1 \text{ in } t_2 : T_2} \qquad \text{(T-LetPoly)}$$

where $\mathcal{X}$ ' are the type variables free in T_1
but not free in Γ

$$\Gamma, x : \forall \mathcal{X}.T \mid\!- \; x : [\mathcal{X} \Rightarrow \mathcal{X}']T \qquad \text{(T-PolyInst)}$$

where $\mathcal{X}$ ' is a set of fresh type variables

let-polymorphism example

double : ∀X. (X -> X) -> X -> X

let double = λf. λx. f(f x)
in let a = double (λx: Nat. succ x) 2
in let b = double (λx: not x) false
in (a,b)

(Y -> Y) -> Y -> Y

(Z -> Z) -> Z -> Z

Then unification yields [Y => Nat, Z => Bool].

let-polymorphism and references

Let ref, !, and := be polymorphic functions with types

```
ref  :  ∀X. X -> Ref(X)
!    :  ∀X. Ref(X) -> X
:=   :  ∀X. Ref(X) *X -> Unit
```

```
let r = ref(λx. x)
in let a = r := (λx: Nat. succ x)
in let b = !r false
in ()
```

r : ∀X. Ref(X -> X)

Ref(Nat -> Nat)

Ref(Bool -> Bool)

We've managed to apply (λx: Nat. succ x) to false!

The value restriction

We correct this unsoundness by only allowing polymorphic generalization at let declarations if the expression is a *value*. This is called the value restriction.

```
let r = ref(λx. x)
in let a = r := (λx: Nat. succ x)
in let b = !r false
in ()
```

r : Ref(X -> X)

Ref(Nat -> Nat) [X => Nat]

Ref(Nat -> Nat)

Now we get a type error in " !r false ".

Let polymorphism with recursive values

Another problem comes when we add recursive value definitions.

let rec f = λx. t in ...

is typed as though it were written

let f = fix(λf. λx. t) in ...

where fix : ∀X. (X -> X) -> X
except that the type of the outer f can be generalized.

Note that the inner f is λ-bound, not let bound, so it cannot be polymorphic within the body t.

Polymorphic Recursion

What can we do about recursive function definitions where the function is polymorphic and is used polymorphically in the body of it's definition? (This is called polymorphic recursion.)

let rec f = λx. (f true; f 3; x)

Have to use a fancier form of type reconstruction: the iterative Mycroft-Milner algorithm.