Hindley Milner Type Inference, Cont.
Announcements

- HW6 (and other HWs) due by the end of term, April 25th

- HW7 is out next week and it is optional

- Choose papers!
Outline

- Hindley Milner type inference
  - Expression and type syntax
  - Instantiations and generalization
  - Typing rules
  - Type inference
    - Strategy 1 or
    - Strategy 2 as known as Algorithm W
  - Observations and examples
Towards Hindley Milner

let f = \lambda x.x

in

if (f true) then (f 1) else 1

- Constraints

  \( t_f = t_1 \rightarrow t_1 \)

  \( t_f = \text{bool} \rightarrow t_2 \) // at call \( f \text{ true} \)

  \( t_f = \text{int} \rightarrow t_3 \) // at call \( f \text{ 1} \)

- Doesn’t unify!
Expression Syntax (to study Hindley Milner)

- Expressions:
  \[ E ::= c \mid x \mid \lambda x. E_1 \mid E_1 E_2 \mid \text{let } x = E_1 \text{ in } E_2 \]

- There are no types in the syntax

- The type of each sub-expression is derived by the **Hindley Milner type inference algorithm**
Type Syntax
(to study Hindley Milner)

- Types (aka monotypes):
  - $\tau ::= b \mid \tau_1 \to \tau_2 \mid t$
  - E.g., int, bool, int $\to$ bool, $t_1 \to$ int, $t_1 \to t_1$, etc.

- Type schemes (aka polymorphic types):
  - $\sigma ::= \tau \mid \forall t.\sigma$
  - E.g., $\forall t_1. \forall t_2. (\text{int} \to t_1) \to t_2 \to t_3$
  - Note: all quantifiers appear in the beginning, $\tau$ cannot contain schemes

- Type environment now

  Gamma ::= Identifiers $\to$ Type schemes
Instantiations

- Type scheme $\sigma = \forall t_1 \ldots t_n.\tau$ can be instantiated into a type $\tau'$ by substituting types for the bound variables ($\text{BV}$) under the universal quantifier $\forall$
  - $\tau' = S \tau$  
  - $S$ is a substitution s.t. $\text{Domain}(S) \supseteq \text{BV}(\sigma)$
  - $\tau'$ is said to be an instance of $\sigma$ ($\sigma > \tau'$)
  - $\tau'$ is said to be a generic instance when $S$ maps some type variables to new type variables
- E.g., $\sigma = \forall t_1.\ t_1 \rightarrow t_2$
Generalization (aka Closing)

- We can generalize a type $\tau$ as follows
  \[
  \text{Gen}(\Gamma, \tau) = \forall t_1, \ldots, t_n. \tau
  \]
  where $\{t_1 \ldots t_n\} = \text{FV}(\tau) - \text{FV}(\Gamma)$

- Generalization introduces polymorphism

- Quantify type variables that are free in $\tau$ but are not free in the type environment $\Gamma$
  - E.g., $\text{Gen}([], t_1 \rightarrow t_2)$ yields
  - E.g., $\text{Gen}([x:t_2], t_1 \rightarrow t_2)$ yields
let $f = \lambda x. x$ in if (f true) then (f 1) else 1

- We’ll infer type for $\lambda x. x$ using simple type inference: $t_1 \rightarrow t_1$

- Then we’ll generalize that type, $\text{Gen}([], t_1 \rightarrow t_1)$:
  \[ \forall t_1. t_1 \rightarrow t_1 \]

- Then we’ll pass the polymorphic type into if (f true) then (f 1) else 1 and instantiate for each f in if (f true) then (f 1) else 1
  - E.g., $[u_2/t_1] (t_1 \rightarrow t_1)$ where $u_2$ is fresh type variable at (f 1)
Generalization, Examples

- \( \lambda f : t_f. \lambda x : t_x. \) let \( g = f \) in \( g\ x \)
  - \( \text{Gen}([f : t_f, x : t_x], t_f) \) yields?

Why can’t we generalize \( t_f \)?

Suppose we can generalize to \( \forall t_f \)
- Then \( \forall t_f = t_g \) will instantiate at \( g\ x \) to some fresh \( u \)
- Then \( u \) becomes \( t_x \rightarrow u' \) thus losing the important connection between \( t_x \) and \( t_f \)!
- Thus \( (\lambda f : t_f. \lambda x : t_x. \) let \( g = f \) in \( g\ x) \) (\( \lambda y . y + 1 \)) \text{ true} \) will type-check (unsound!!!)

DO NOT generalize variables that are mentioned in type environment \( \Gamma \)!
Hindley Milner Typing Rules

\[
\Gamma; x: \tau |- E_1 : \tau \quad \Gamma; x: \text{Gen}(\Gamma, \tau) |- E_2 : \tau' \\
\Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau'
\]

(Let)

- Type of \( x \) as inferred for \( E_1 \) is \( \tau \). Type of \( x \) in \( E_2 \) is the generalized type scheme \( \sigma = \text{Gen}(\Gamma, \tau) \)

\[
x: \sigma \in \Gamma \quad \tau < \sigma \quad (\text{Var})
\]

\[
\Gamma |- x : \tau
\]

- \( x \) in \( E_2 \) of \text{let}: \( x \) is of type \( \tau \) if its type \( \sigma \) in the environment can be instantiated to \( \tau \)

(Note: remaining rules, \( c, \text{App, Abs} \) are as in \( F_1 \).)
Outline

- Hindley Milner type inference
  - Expression and type syntax
  - Instantiations and generalization
  - Typing rules
  - Type inference
    - Strategy 1 or
    - Strategy 2 as known as Algorithm W
  - Observations and examples
let \( x = E_1 \) in \( E_2 \)

1. Calculate type \( T_{E_1} \) for \( E_1 \) in \( \Gamma; x: t_x \) using simple type inference

2. Generalize free type variables in \( T_{E_1} \) to get the type scheme for \( T_{E_1} \) (be mindful of caveat!)

3. Extend environment with \( x: \text{Gen}(\Gamma, T_{E_1}) \) and start typing \( E_2 \)

4. Every time we encounter \( x \) in \( E_2 \), instantiate its type scheme using fresh type variables

E.g., \( \text{id} \)'s type scheme is \( \forall t_1. t_1 \rightarrow t_1 \) so \( \text{id} \) is instantiated to \( u_k \rightarrow u_k \) at \( \text{(id 1)} \)
Hindley Milner Type Inference

- Two ways:
  - Extend Strategy 1 (constraint-based typing)
  - Extend Strategy 2 (Algorithm W)
Strategy 1

\[ \text{let } f = \lambda x \cdot x \text{ in if (f true) then (f 1) else 1} \]

1. let \( \Gamma = [] \)
   \( t_1 = t_3 \)
   \( \Gamma = [f : t_f] \)

2. Abs
   \( t_2 = t_x \rightarrow t_x \)
   \( \Gamma = [f : t_f, x : t_x] \)

3. if-then-else
   \( t_3 = t_5 = \text{int} \)
   \( t_4 = \text{bool} \)

4. App
   \( u_1 \rightarrow u_1 = \text{bool} \rightarrow t_4 \)
   \( u_2 \rightarrow u_2 = \text{int} \rightarrow t_5 \)

5. App
   \( f \)
   \( \text{true} \)
   \( f \)
   \( 1 \)

Next, generalize \( t_f : \forall t_x \cdot t_x \rightarrow t_x \)

\( u_1 \) and \( u_2 \) are fresh type vars generated at instantiation of polymorphic type.
Example

\[ \lambda x. \text{let } f = \lambda y. x \text{ in } (f \text{ true}, f \text{ 1}) \]
def W(Γ, E) = case E of
  c   -> ([], TypeOf(c))
  x   -> if (x NOT in Domain(Γ)) then fail
       else let T_E = Γ(x)
         in case T_E of
            _ -> ([], T_E)
            τ -> ([], [u_1/t_1...u_n/t_n] τ )
  λx.E_1 -> let (S_{E_1}, T_{E_1}) = W(Γ+{x:t_x},E_1)
          in (S_{E_1}, S_{E_1}(t_x)→T_{E_1})

  // ...

  // continues on next slide!
Strategy 2: Algorithm W

def W(Γ, E) = case E of

    // continues from previous slide
    // ...
    E₁ E₂  -> let (Sₑ₁,Tₑ₁) = W(Γ,E₁)
              (Sₑ₂,Tₑ₂) = W(Sₑ₁(Γ),E₂)
              S = Unify(Sₑ₂(Tₑ₁),Tₑ₂→t)
        in (S Sₑ₂ Sₑ₁, S(t))

    let x = E₁ in E₂  -> let (Sₑ₁,Tₑ₁) = W(Γ+{x:tₓ},E₁)
                                  S = Unify( Sₑ₁(tₓ),Tₑ₁ )
                                  σ = Gen( S Sₑ₁(Γ), S(Tₑ₁) )
              (Sₑ₂,Tₑ₂) = W(S Sₑ₁(Γ)+{x:σ},E₂)
        in (Sₑ₂ S Sₑ₁, Tₑ₂)
let f = \( \lambda x . x \) in if (f true) then (f 1) else 1

1. let \( \Gamma = [f : t_f] \)  \( T_1 = \text{int} \)  \( S_1 = \ldots \)  \( \Gamma = [f : \forall t_x . t_x \rightarrow t_x] \)  \( \Gamma = \emptyset \)  \( T_2 = t_x \rightarrow t_x \)  \( S_2 = \emptyset \)  \( \Gamma = [x : t_x, f : t_f] \)

2. Abs

\[ f : t_x \rightarrow t_x \]

\[ S_2 = \emptyset \]

No constraint, types immediately: \( T_2 = t_x \rightarrow t_x : [t_x \rightarrow t_x / t_2] \)

\( \sigma = \text{Gen}(\emptyset, t_x \rightarrow t_x) = \forall t_x . t_x \rightarrow t_x \)

3. if-then-else  \( \Gamma = \emptyset \)  \( T_3 = \text{int} \)  \( S_3 = \ldots \)  \( f : \forall t_x . t_x \rightarrow t_x \)

4. App

\[ T_4 = \text{bool} \]

\[ S_4 = [\text{bool} / t_4][\text{bool} / u_1] \]

\[ T_5 = \text{int} \]

\[ S_5 = [\text{int} / t_5][\text{int} / u_2] \]

5. App

\[ T = u_1 \rightarrow u_1 \]

\[ S = \emptyset \]

From \( \text{Unify}(u_1 \rightarrow u_1, \text{bool} \rightarrow t_4) \)
Hindley Milner Observations

- Do not generalize over type variables mentioned in type environment (they are used elsewhere)

- `let` is the only way of defining polymorphic constructs

- Generalize the types of let-bound identifiers **only after** processing their definitions
Hindley Milner Observations

- Generates the **most general type** (principal type) for each term/subterm
- Type system is sound

- Complexity of Algorithm W
  - PSPACE-Hard
  - Because of nested let blocks
Hindley Milner Limitations

- Only let-bound constructs can be polymorphic and instantiated differently

```verbatim
let twice f x = f (f x)
in twice twice succ 4 // let-bound polymorphism
```

```verbatim
let twice f x = f (f x)
  foo g = g g succ 4 // lambda-bound
in foo twice
```
Hindley Milner Limitations

Quiz example:

\[(\lambda x. x (\lambda y. y) (x \ 1)) (\lambda z. z)\]

vs.

let x = (\lambda z. z)

in

x (\lambda y. y) (x \ 1)