Announcements

- HW6 on Submitty
- Presentation guidelines up on Schedule page
  1. Select available paper/slot from list (first-come-first-serve)
  2. If available, I’ll assign and update, otherwise goto 1

Quiz 5

Outline

- Hindley Milner type inference
- Algorithm W
- Hindley Milner observations

Towards Hindley Milner

let f = \( \lambda x.x \) in
if (f true) then (f 1) else 1

Constraints
- \( t_f = t_f_1 \rightarrow t_1 \)
- \( t_f = \text{bool} \rightarrow t_2 \) // at call (f true)
- \( t_f = \text{int} \rightarrow t_3 \) // at call (f 1)

Doesn’t unify!

Towards Hindley Milner

- Extends simple type inference with let polymorphism
- let \( f = \lambda x.x \) // Infer polymorphic type for f: \( \forall T. T \rightarrow T \)
in if (f true) then (f 1) else 1

- At call (f true), \( t_f \) instantiates to \( u_1 \rightarrow u_1 \) and \( u_1 \rightarrow u_1 \) unifies with \( \text{bool} \rightarrow t_2 \)
- At call (f 1), \( t_f \) instantiates to \( u_2 \rightarrow u_2 \) and \( u_2 \rightarrow u_2 \) unifies with \( \text{int} \rightarrow u_3 \)

Importantly, \( u_1 \) and \( u_2 \) are fresh variables

Towards Hindley Milner

- Solution:
  - Generalize the type variable in type of f
    - \( t_f : t_f_1 \rightarrow t_1 \) becomes \( t_f : \forall T. T \rightarrow T \)
  - Different uses of generalized type variables are instantiated differently
    - E.g., (f true) instantiates \( t_f \) into \( \text{bool} \rightarrow \text{bool} \)
    - E.g., (f 1) instantiates \( t_f \) into \( \text{int} \rightarrow \text{int} \)
  - When can we generalize?
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):
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t \]
  \( t \) is a type variable
- E.g., int, bool, int\rightarrow bool, t_1\rightarrow int, t_1t_2, etc.
- Type schemes (aka polymorphic types):
  \[ \sigma ::= \tau \mid \forall \tau \sigma \]
  E.g., \( \forall \tau \), \( \forall \tau \)(\(\text{int}\rightarrow \tau_1\rightarrow \tau_2\rightarrow \tau_3\))
  \( \sigma \) is a "free" type variable as it isn't bound under \( \forall \)
- Note: all quantifiers appear in the beginning, \( \tau \) cannot contain schemes
- Type environment Gamma now

Generalization (aka Closing)
- We can generalize a type \( \tau \) as follows
  \[ \text{Gen}(\Gamma, \tau) = \forall t_1 \ldots t_n \tau \]
  where \( \{t_1 \ldots t_n\} = \text{FV}(\tau) \setminus \text{FV}(\Gamma) \)
- Generalization introduces polymorphism!
- Quantify type variables that are free in \( \tau \) but are not free in the type environment \( \Gamma \)
  - E.g., \( \text{Gen}(\Gamma, t_1\rightarrow t_2) \) yields \( \forall t_1 \rightarrow t_2 \)
  - E.g., \( \text{Gen}(\Gamma, x:t) \) yields \( \forall t \)

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 (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 \( \sigma \) (\( \sigma \succ \tau' \))
  \( \tau' \) is said to be a generic instance when \( S \) maps some type variables to new type variables
- E.g., \( \sigma = \forall t_1, t_2 \rightarrow t_3 \)
  \[ [t_2/t_1] t_1 \rightarrow t_2 = t_2 \rightarrow t_3 \] is a generic instance of \( \sigma \)
  \[ [\text{int/t_1}] t_1 \rightarrow t_3 = \text{int} \rightarrow t_3 \] is a non-generic instance of \( \sigma \)

Generalization, Examples
- 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
- \( \lambda x.x \) : \( \tau \). let \( g = f \) in \( g \ x \)
  - \( \text{Gen}([f; t_n; x]; t_1 \rightarrow t_1) \) yields?
  - Why can't we generalize \( t_1 \)?: \( \forall \text{Gen} \)
  - Suppose we can generalize to \( \forall t_n t_1 \)
    - Then \( \forall t_1, t_n = t_1 \) will instantiate at \( g \ x \) to some fresh \( u \)
    - Then \( u \) unifies with \( t_1 \rightarrow u' \) thus losing the important connection between \( t_1 \) and \( t_n \). \( t_1 \rightarrow t_n \rightarrow \ldots \)
    - Thus \( (f t_n; \lambda x.x); \text{let } g=f\text{ in } g \ x \) \((y y+1)\) true will type-check (unsound!!!)
  - DO NOT generalize variables that are mentioned in type environment \( \Gamma \)!
Hindley Milner Typing Rules

\[ \frac{\Gamma; \tau \vdash E_1 : \tau \quad \Gamma; \text{Gen}(\Gamma; \tau) \vdash E_2 : \tau}{\Gamma \vdash \text{let } x = E_2 \in E_1 : \tau} \]  
(Left)

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

\[ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \]  
(var)

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

(Note: remaining rules, c, App, Abs are as in \( F_\tau \))

---

Hindley Milner Type Inference, Rough Sketch

let \( x \) = \( E_1 \) in \( E_2 \)

1. Calculate type \( T_{E_1} \) for \( E_1 \) in \( \Gamma; x : \tau \)
2. Generalize free type variables in \( T_{E_1} \) to get the type scheme for \( T_{E_1} \) (be mindful of caveats!)
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., \( id \)'s type scheme is \( \forall t_1 . t_2 \rightarrow t_3 \) so \( id \) is instantiated to \( u_k \rightarrow u_k \) at (id 1)

---

Strategy 2: Algorithm W

def \( W(\Gamma; E) = \text{case } E \) of

\[ c \rightarrow ([\], \text{TypeOf}(c)) \]
\[ x \rightarrow \text{if } (x \text{ NOT IN Domain}(\Gamma)) \text{ then fail } \]
\[ \text{else let } T_x = \Gamma(x) \text{ in case } T_x \text{ of } \]
\[ \forall t_1 \ldots t_k \rightarrow ([u_1 . \ldots . u_n . \tau] \tau) \]
\[ \rightarrow ([\], \tau) \]
\[ \lambda x . E_1 \rightarrow \text{let } (S_{E_1}, T_{E_1}) = W(\Gamma; (x : \tau), E_1) \]
\[ \text{in } (S_{E_1}, S_{E_1}, T_{E_1}) \]

if ... 
// continues on next slide!

---

Strategy 2 Example

let \( f = \lambda x . x \) in

if \( (f \text{ true}) \) then \( (f 1) \) else 1

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

2. Abs

\[ \Gamma = [t_1 . t_2 \rightarrow t_3] \]
\[ T_1 = \text{int} \]
\[ S_1 = [\] \]
\[ T_2 = t_1 . t_2 \rightarrow t_3 \]
\[ S_2 = [\] \]
\[ T_3 = \text{int} \]
\[ S_3 = [\] \]
\[ \lambda x : t_2 . x \]
\[ T_x = t_2 \rightarrow t_3 \]
\[ S_x = [\] \]

No constraint, types 2. Abs immediately: \( T_1 = t_1 . t_2 \rightarrow t_3 \)
\( \Rightarrow \)
\( \gamma = \text{Gen}([t_1 . t_2 . t_3] = [\gamma] \)

\[ \Gamma = [t_1 . t_2 . t_3] \]
\[ T_1 = \text{int} \]
\[ S_1 = [\] \]
\[ T_2 = t_1 . t_2 \rightarrow t_3 \]
\[ S_2 = [\] \]
\[ T_3 = \text{int} \]
\[ S_3 = [\] \]

4. App

\[ f \]
\[ T_f = \text{bool} \]
\[ S_f = [\text{bool}, \text{int}, \text{int}, \text{int}] \]

5. App

\[ f \]
\[ T_f = \text{bool} \]
\[ S_f = [\text{bool}, \text{int}, \text{int}, \text{int}] \]

From \( \text{Unify}(u_k, u_k, \text{bool}, t_3) \)
Example

\[ \lambda x. \text{let } f = \lambda y. x \text{ in } (f \text{ true}, f \text{ 1}) \]

Hindley Milner Observations

- Do not generalize over type variables mentioned in type environment (they are used elsewhere)
- \textit{let} is the only way of defining polymorphic constructs
- Generalize the types of let-bound identifiers \textbf{only after} processing their definitions

Hindley Milner Observations

- Generates the \textbf{most general type (principal type)} for each term/subterm
- Type system is sound
- Complexity of Algorithm W
  - \textit{PSPACE-Hard}
  - Because of nested let blocks

Hindley Milner Limitations

- Quiz example:
  \[ (\lambda x. (\lambda y. (x \text{ 1}))) (\lambda z. z) \]
  vs.
  \[ \text{let } x = (\lambda z. z) \text{ in } x (\lambda y. (x \text{ 1})) \]

Hindley Milner Limitations

- Only let-bound constructs can be polymorphic and instantiated differently
  \texttt{let twice f x = f (f x) in twice twice succ 4 // let-bound polymorphism}

- \texttt{let twice f x = f (f x)}
  - foo g = g g succ 4 // lambda-bound
  - in foo twice

The End
\[ S = \lambda x. \lambda y. \lambda z. \ x \ z \ (y \ z) \]