Hindley Milner Type Inference
Announcements

- HW6?
- Presentation guidelines are up, papers are up on schedule page as well
  - 1. Select available paper/slot from list
  - 2. If available, I’ll assign and update, otherwise goto 1.

- 3 broad topics
  - ML in Program Analysis
  - Security: Binary analysis and Obfuscation
  - Dynamic Binary Instrumentation (DBI)
Outline

- Simple type inference, conclusion
  - Let constructs
  - Strategy 2: on-the-fly typing
- Parametric polymorphism
- Hindley Milner type inference. Algorithm W
Type Inference

Strategy 1 solves constraints offline
- Use typing rules to generate type constraints
- Solve type constraints “offline”
- Essential concepts: equality, unification and substitution

Strategy 2 solves constraints on the fly
- Builds the substitution map incrementally
The Let Construct

- In dynamic semantics, \( \textbf{let } x = E_1 \textbf{ in } E_2 \) is equivalent to \( (\lambda x. E_2) \ E_1 \)

Typing rule

\[
\Gamma |- E_1 : \sigma \quad \Gamma;x:\sigma |- E_2 : \tau
\]

\[
\Gamma |- \textbf{let } x = E_1 \textbf{ in } E_2 : \tau
\]

- In static semantics \( \textbf{let } x = E_1 \textbf{ in } E_2 \) is not equivalent to \( (\lambda x. E_2) \ E_1 \)
  - In \textbf{let}, the type of “argument” \( E_1 \) is inferred/checked \textbf{before} the type of function body \( E_2 \)
  - \textbf{let} construct enables Hindley Milner style polymorphism!
The Let Construct

- Typing rule

\[ \Gamma \vdash E_1 : \sigma \quad \Gamma ; x : \sigma \vdash E_2 : \tau \]

\[ \Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau \]

- Attribute grammar rule

\[ E ::= \text{let } x = E_1 \text{ in } E_2 \]

\[ \Gamma_{E_1} = \Gamma_E \]

\[ \Gamma_{E_2} = S_{E_1}(\Gamma_E) + \{ x : T_{E_1} \} \]

\[ T_E = T_{E_2} \quad S_E = S_{E_2} S_{E_1} \]
Typing Let Terms

\[ \text{let } x = \frac{1}{\text{in }} \frac{x + 1}{\text{in int}} \]
\[ \text{Nil } \vdash \text{let } \ldots \text{ in int} \]

\[ \text{let } \text{twice } = \lambda f. \lambda x. f(f(x)) \text{ in twice } (\lambda x. x + 1) \ 2 \]
\[ \text{Nil } \vdash \text{let } \ldots \text{ in int} \]

\[ \text{let } \text{plus } = \lambda x. \lambda y. \text{if } x = 0 \text{ then } y \text{ else plus } (x - 1) \ (y + 1) \text{ in plus } 2 \ 3 \]

\[ \text{Nil } \vdash \text{let } \ldots \text{ in bool} \]

\[ \text{ERROR} \]
The Letrec Construct

- letrec x = E₁ in E₂
  - x can be referenced from within E₁
  - Extends calculus with general recursion
    - No need to type fix (we can’t!) but we can still type recursive functions like plus, times, etc.
  - Haskell’s let is a letrec actually!

- E.g.,
  letrec plus = λx.λy. if (x=0) then y else ((plus x-1) y+1) in ...
or in Haskell syntax:
  let plus x y = if (x=0) then y else plus (x-1) (y+1) in ...
The Letrec Construct

- **letrec** \( x = E_1 \) in \( E_2 \)

**Attribute grammar rule**

\[
E ::= \text{letrec } x = E_1 \text{ in } E_2
\]

Extensions over let rule

1. \( T_{E_1} \) is inferred in augmented environment \( \Gamma_E + \{x:t_x\} \)
2. Must unify \( S_{E_1}(t_x) \) and \( T_{E_1} \)
3. Apply substitution \( S \) on top of \( S_{E_1} \)

Note: Can merge let and letrec, in let

**Unify** and \( S \) have no impact
let vs. letrec

let \( \text{plus} = \lambda x. \lambda y. \text{if } (x=0) \text{ then } y \text{ else } ((\text{plus } x-1) y+1) \) in

... LET

\( \Gamma = [] \)

\( E_i = \begin{cases} \text{error} & \text{if } \text{plus } x-1 \notin \Gamma \\ \text{error} & \text{if } \text{plus } y+1 \notin \Gamma \\ \text{else } \text{letrec } \end{cases} \)

\( \text{LETREC} \)

\( E_1 \) let

\( E_1 = \begin{cases} \text{error} & \text{if } \text{plus } x-1 \notin \Gamma \\ \text{error} & \text{if } \text{plus } y+1 \notin \Gamma \\ \text{else } \text{letrec } \end{cases} \)

\( \Gamma = [\text{plus : t\text{plus}}] \)

\( \Gamma = [\text{plus : t\text{plus}}] \)

\( \text{t\text{plus}} = \text{iut} \rightarrow \text{t\text{plus}} \rightarrow \text{t\text{plus}} \)

2. App

3. App

\( \text{t\text{plus}} = \text{iut} \rightarrow \text{t_2} \)

\( \text{t\text{plus}} = \text{iut} \rightarrow \text{t_3} \)

\( \text{t\text{plus}} = \text{iut} \)

\( \text{t\text{plus}} \text{ x-1 iut} \)
Algorithm W, Almost There!

def W(Γ, E) = case E of
   c    -> ([], TypeOf(c))
   x    -> if (x NOT in Dom(Γ)) then fail
          else let T_E = Γ(x);
          in ([], T_E)
   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})
   E_1 E_2 -> let (S_{E_1}, T_{E_1}) = W(Γ,E_1)
              (S_{E_2}, T_{E_2}) = W(S_{E_1}(Γ), E_2)
              t_{E_1} = t_{E_2} \rightarrow t_E
              S = Unify(S_{E_2}(T_{E_1}), T_{E_2} \rightarrow t)
              in (S S_{E_2} S_{E_1}, S(t)) // S S_{E_2} S_{E_1} composes substitutions
   let x = E_1 in E_2 -> let (S_{E_1}, T_{E_1}) = W(Γ,E_1)
                           (S_{E_2}, T_{E_2}) = W(S_{E_1}(Γ)+{x:T_{E_1}},E_2)
                           in (S_{E_2} S_{E_1}, T_{E_2})
Algorithm W, Almost There!
(merges let and letrec)

def W(Γ, E) = case E of
    c   ->  ([], TypeOf(c))
    x   ->  if (x NOT in Dom(Γ)) then fail
             else let T_E = Γ(x);
                in ([], T_E)
    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})
    E_1 E_2 -> let (S_{E_1}, T_{E_1}) = W(Γ,E_1)
              (S_{E_2}, T_{E_2}) = W(S_{E_1}(Γ), E_2)
              S = Unify(S_{E_2}(T_{E_1}), T_{E_2}→t)
              in (S S_{E_2} S_{E_1}, S(t)) // S S_{E_2} S_{E_1} composes substitutions
    let x = E_1 in E_2 -> let (S_{E_1}, T_{E_1}) = W(Γ+{x:t_x}, E_1)
                           S = Unify(S_{E_1}(t_x), T_{E_1})
                           (S_{E_2}, T_{E_2}) = W(S S_{E_1}(Γ)+{x:T_{E_1}}, E_2)
                           in (S_{E_2} S S_{E_1}, T_{E_2})
Outline

- Simple type inference, conclusion
  - Let constructs
  - Strategy 2: on-the-fly typing
- Parametric polymorphism
  - Hindley Milner type inference. Algorithm W
Motivating Example

- A sound type system rejects some programs that don’t get stuck.
- Canonical example

\[ \text{let } f = \lambda x. x \]

in

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

- Term does not get “stuck”
- Term is NOT TYPABLE in the simply typed lambda calculus. It is typable in Hindley Milner!
Different Styles of (Parametric) Polymorphism

- Impredicative polymorphism (System F)

\[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \mid \forall T. \tau \]

\[ E ::= x \mid \lambda x: \tau. E \mid E_1 E_2 \mid \Gamma T. E \mid E[\tau] \]

- Very powerful
  - Can type self application \((\lambda x. x x)\)
  - Still cannot type \(\text{fix!}\)

- Type inference is undecidable!
Different Styles of Polymorphism

- Predicative polymorphism
  - $\tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T$
  - $\sigma ::= \tau \mid \forall T.\sigma \mid \sigma_1 \rightarrow \sigma_2$
  - $E ::= x \mid \lambda x:\sigma. E \mid E_1 E_2 \mid \Lambda T.E \mid E[\tau]$

- Still very powerful
  - Restricts System F by disallowing instantiation with a polymorphic type: $E[\tau]$ but not $E[\sigma]$
  - Type inference is still undecidable!
Different Styles of Polymorphism

- Prenex polymorphism
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall T. \sigma \]
  \[ E ::= x \mid \lambda x : \tau. E \mid E_1 E_2 \mid \Lambda T. E \mid E[\tau] \]

- Now type inference is decidable
- But polymorphism is limited
  - You cannot pass polymorphic functions
  - E.g., we cannot pass a sort function as argument
Different Styles of Polymorphism

- Let polymorphism

\[
\tau ::= \text{b} \mid \tau_1 \rightarrow \tau_2 \mid T
\]

\[
\sigma ::= \tau \mid \forall T.\sigma
\]

\[
E ::= x \mid \lambda x: \tau. E \mid E_1 E_2 \mid \Lambda T. E \mid E[\tau] \mid \text{let x = E}_1 \text{ in } E_2
\]

- Like \((\lambda x. E_2) E_1\) but \(x\) can be polymorphic!

- Good engineering compromise
  - Enhance expressiveness
  - Preserve decidability

- This is the Hindley Milner type system
Outline

- Simple type inference, conclusion
  - Let constructs
  - Strategy 2: on-the-fly typing

- Parametric polymorphism

- Hindley Milner type inference. Algorithm W
Towards Hindley Milner

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

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

Doesn’t unify!
Solution:

Generalize the type variable in type of $f$

$t_f : t_1 \rightarrow t_1$ becomes $t_f : \forall T. T \rightarrow T$

Different uses of generalized type variables are instantiated differently

- E.g., $(f \text{ true})$ 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: $E ::= c | x | \lambda x.E_1 | E_1 E_2 | \text{let } x = E_1 \text{ in } E_2$

- 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):
- \( \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t \)
- E.g., int, bool, int \rightarrow bool, t_1 \rightarrow int, t_1 \rightarrow t_1, \text{etc.}

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

Type environment now

Gamma ::= Identifiers \rightarrow 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 (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., $\sigma = \forall t_1. t_1 \rightarrow t_2$
  - $[t_3/t_1] t_1 \rightarrow t_2 = t_3 \rightarrow t_2$ is a generic instance of $\sigma$
  - $[\text{int}/t_1] t_1 \rightarrow t_2 = \text{int} \rightarrow t_2$ is a non-generic instance of $\sigma$
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 $\forall t_1, t_2. t_1 \rightarrow t_2$
  - E.g., $\text{Gen}([x, t_2], t_1 \rightarrow t_2)$ yields $\forall t_1. t_1 \rightarrow t_2$
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, 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)