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, \( \text{let } x = E_1 \text{ in } E_2 \) is equivalent to \( (\lambda x. E_2) \ E_1 \)
- Typing rule
  \[
  \Gamma |- E_1 : \sigma \quad \Gamma; x: \sigma |- E_2 : \tau
  \]
  \[
  \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau
  \]
- In static semantics \( \text{let } x = E_1 \text{ in } E_2 \) is not equivalent to \( (\lambda x. E_2) \ E_1 \)
  - In let, the type of “argument” \( E_1 \) is inferred/checked before the type of function body \( E_2 \)
  - let construct enables Hindley Milner style polymorphism!
The **Let Construct**

- **Typing rule**

\[
\begin{align*}
\Gamma |- E_1 : \sigma & \quad \Gamma;\!x:σ |- E_2 : \tau \\
\Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau
\end{align*}
\]

- **Attribute grammar rule**

\[
E ::= \text{let } x = E_1 \text{ in } E_2 \\
\Gamma_{E_1} = \Gamma_E \\
\Gamma_{E_2} = S_{E_1}(\Gamma_E) + \{x:T_{E_1}\} \\
T_E = T_{E_2} \\
S_E = S_{E_2} S_{E_1}
\]

Program Analysis CSCI 4450/6450, A Milanova
Typing Let Terms
The Letrec Construct

- \textbf{letrec} \( x = E_1 \) in \( E_2 \)
  - \( x \) can be referenced from within \( E_1 \)
  - Extends calculus with general recursion
    - No need to type \textbf{fix} (we can’t!) but we can still type recursive functions like \texttt{plus}, \texttt{times}, etc.
  - Haskell’s \texttt{let} is a \texttt{letrec} actually!

- E.g.,

\begin{align*}
\text{letrec } \texttt{plus} &= \lambda x. \lambda y. \text{ if } (x=0) \text{ then } y \text{ else } ((\texttt{plus} \ x - 1) \ y + 1) \text{ in } \\
\text{or in Haskell syntax:} \quad \texttt{let } \texttt{plus} \ x \ y &= \text{ if } (x=0) \text{ then } y \text{ else } \texttt{plus} \ (x-1) \ (y+1) \text{ in } \\
\end{align*}
The Letrec Construct

- letrec x = E₁ in E₂

---

Attribute grammar rule

E ::= letrec x = E₁ in E₂

\[ \Gamma_{E₁} = \Gamma_{E} + \{x : t_x\} \]
\[ S = \text{Unify}(S_{E₁}(t_x), T_{E₁}) \]
\[ \Gamma_{E₂} = S \ S_{E₁}(\Gamma_{E}) + \{x : T_{E₁}\} \]
\[ T_{E} = T_{E₂} \quad S_{E} = S_{E₂} \ S \ S_{E₁} \]
let vs. letrec

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

...
Algorithm W, Almost There!

```python
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,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)

```python
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_E1, T_E1) = W(Γ+{x:t_x}, E_1)
        in (S_E1, S_E1(t_x)→T_E1)
    E_1 E_2 -> let (S_E1, T_E1) = W(Γ, E_1)
        (S_E2, T_E2) = W(S_E1(Γ), E_2)
        S = Unify(S_E2(T_E1), T_E2→t)
        in (S S_E2 S_E1, S(t)) // S S_E2 S_E1 composes substitutions
    let x = E_1 in E_2 -> let (S_E1, T_E1) = W(Γ+{x:t_x}, E_1)
        S = Unify(S_E1(t_x), T_E1)
        (S_E2, T_E2) = W(S S_E1(Γ)+{x:T_E1}, E_2)
        in (S_E2 S S_E1, T_E2)
```

Program Analysis CSCI 4450/6450, A Milanova (from MIT 2015 Program Analysis OCW)
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
  
  \[
  \text{let } f = \lambda x.x \\
  \text{in} \\
  \text{if } (f \text{ true}) \text{ then } (f \text{ 1}) \text{ else 1}
  \]

  - 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)
  \[ \tau ::= b \mid \tau_1 \to \tau_2 \mid T \mid \forall T.\tau \]
  \[ E ::= x \mid \lambda x:\tau. E \mid E_1 E_2 \mid \Lambda T. E \mid E[\tau] \]
  Can instantiate with polymorphic type!

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

- Type inference is undecidable!
Different Styles of Polymorphism

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

- 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
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall T. \sigma \]
  \[ E ::= x \mid \lambda x : \tau. E \mid E_1 E_2 \mid \Lambda T. E \mid E[\tau] \mid \text{let } x = E_1 \text{ in } E_2 \]

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

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

- **Constraints**

\[
\begin{align*}
\text{t}_f &= \text{t}_1 \rightarrow \text{t}_1 \\
\text{t}_f &= \text{bool} \rightarrow \text{t}_2 \quad // \text{at call (f true)} \\
\text{t}_f &= \text{int} \rightarrow \text{t}_3 \quad // \text{at call (f 1)}
\end{align*}
\]

- 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., \( f \text{ 1} \) 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
Types (aka monotypes):
- \( \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t \)
- E.g., int, bool, int\( \rightarrow \)bool, \( t_1 \rightarrow \)int, \( t_1 \rightarrow t_1 \), 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

\( \Gamma ::= \text{Identifiers} \rightarrow \text{Type schemes} \)
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., \( \sigma = \forall t_1. t_1 \to t_2 \)
  - \( [t_3/t_1] t_1 \to t_2 = t_3 \to t_2 \) is a generic instance of \( \sigma \)
  - \( [\text{int}/t_1] t_1 \to t_2 = \text{int} \to t_2 \) 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 $\forall t_1. t_1 \rightarrow t_2$
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 \( g \ x \) \) (\( \lambda y . y + 1 \) true) will type-check (unsound!!!)

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

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

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

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

- \( x \) 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, \( c, \text{App}, \text{Abs} \) are as in \( F_1 \).)
let x = E₁ in E₂

1. Calculate type $T_{E₁}$ for $E₁$ in $\Gamma ; x : t_x$ using simple type inference
2. Generalize free type variables in $T_{E₁}$ to get the type scheme for $T_{E₁}$ (be mindful of caveat!)
3. Extend environment with $x : \text{Gen}(\Gamma , T_{E₁})$ and start typing $E₂$
4. Every time we encounter $x$ in $E₂$, instantiate its type scheme using fresh type variables

E.g., id’s type scheme is $\forall t₁.t₁→t₁$ so id is instantiated to $u_k→u_k$ at (id 1)
Hindley Milner Type Inference

- Two ways:
  - Extend Strategy 1 (constraint-based typing)
  - Extend Strategy 2 (Algorithm W)
let f = \(x . x\) in if (f true) then (f 1) else 1

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

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

3. if-then-else
   \(\Gamma = [f : \forall t_x . t_x \rightarrow t_x]\)
   \(t_3 = t_5 = \text{int}\)
   \(t_4 = \text{bool}\)

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 \(t_f : \forall t_x . t_x \rightarrow t_x\)

\(u_1\) 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 \( W(\Gamma, E) = \) case \( E \) of

\begin{align*}
\text{c} & \rightarrow ([], \text{TypeOf(c)}) \\
\text{x} & \rightarrow \text{if (x NOT in Domain(\Gamma)) then fail} \\
& \quad \text{else let } T_E = \Gamma(x) \\
& \quad \text{in case } T_E \text{ of} \\
& \quad \quad \forall \ t_1,...,t_n.\tau \rightarrow ([], [u_1/t_1...u_n/t_n] \tau) \\
& \quad \quad \_ \rightarrow ([], T_E) \\
\lambda x. E_1 & \rightarrow \text{let } (S_{E_1}, T_{E_1}) = W(\Gamma + \{x:t_x\}, E_1) \\
& \quad \text{in } (S_{E_1}, S_{E_1}(t_x) \rightarrow T_{E_1})
\end{align*}

// ...

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

```python
def W(Γ, E) = case E of
    // continues from previous slide
    // ...
    E₁ E₂ -> let (S₁, T₁) = W(Γ, E₁)
    (S₂, T₂) = W(S₁(Γ), E₂)
    S = Unify(S₂(T₁), T₂ → t)
    in (S S₂ S₁, S(t))

    let x = E₁ in E₂ -> let (S₁, T₁) = W(Γ+x::t, E₁)
    S = Unify( S₁(t), T₁ )
    σ = Gen( S S₁(Γ), S(T₁) )
    (S₂, T₂) = W(S S₁(Γ)+x::σ, E₂)
    in (S₂ S₁(Γ)+x::σ, T₂)
```

Program Analysis CSCI 4450/6450, A Milanova (from MIT 2015 Program Analysis OCW)
Strategy 2 Example

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

1. let \( \Gamma = [] \)
   \( S_1 = ... \)
   \( T_1 = \text{int} \)

2. Abs
   \( \Gamma = [] \)
   \( T_2 = t_x \rightarrow t_x \)
   \( S_2 = [] \)
   \( \Gamma = [x : t_x] \)

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

4. App
   \( T_4 = \text{bool} \)
   \( S_4 = [\text{bool} / t_4][\text{bool} / u_1] \)

5. App
   \( T_5 = \text{int} \)
   \( S_5 = [\text{int} / t_5][\text{int} / u_2] \)

No constraint, types

immediately: \( T_2 = t_x \rightarrow t_x : [t_x \rightarrow t_x / t_2] \)
\( \sigma = \text{Gen}([], t_x \rightarrow t_x) = \forall t_x. t_x \rightarrow t_x \)

\( T = u_1 \rightarrow u_1 \)
\( S = [] \)

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

- `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

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

```plaintext
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.

\textbf{let } x = (\lambda z. \; z) \\
\textbf{in} \\
x (\lambda y. \; y) (x \; 1)