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 (first-come-first-serve)
  2. If available, I’ll assign and update, otherwise goto 1.
Outline

- Simple type inference, conclusion
  - 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 and unification

- Strategy 2 solves constraints on the fly
  - Builds the substitution map incrementally
Add a New Attribute, Substitution Map \( S \)

**Grammar rule:**

\[
E ::= x \\
E ::= c \\
E ::= \lambda x.E_1 \\
E ::= E_1 E_2
\]

**Attribute rule:**

\[
T_E = \Gamma_E(x) \quad S_E = [ ] \\
T_E = \text{int} \quad S_E = [ ] \\
\Gamma_{E_1} = \Gamma_E ; x : t_x \\
T_E = S_{E_1}(t_x) \rightarrow T_{E_1} \quad S_E = S_{E_1} \\
\Gamma_{E_1} = \Gamma_E \quad \Gamma_{E_2} = S_{E_1}(\Gamma_E) \\
S = \text{Unify}(S_{E_2}(T_{E_1}), T_{E_2} \rightarrow t_E) \\
T_E = S(t_E) \quad S_E = S \ S_{E_2} \ S_{E_1}
\]

\( T_E \) is the inferred type of \( E \).
\( S_E \) is the substitution map resulting from inferring \( T_E \).
\( t_x, t_E \) are fresh type variables.
Example: \((\lambda f \cdot f \, 5) \, (\lambda x \cdot x)\)

\((\lambda f \cdot f \, 5) \, (\lambda x \cdot x) : ?\)

1. App
   - \(\Gamma_1 = []\)
   - \(T_1 = \text{int}\)
   - \(S_1 = [\text{int/t}_1] [\text{int/t}_3] [\text{int/t}_x] [\text{int->t}_3/t_f]\)

2. Abs
   - \(\Gamma_2 = []\)
   - \(T_2 = (\text{int->t}_3) \rightarrow \text{t}_3\)
   - \(S_2 = [\text{int->t}_3/t_f]\)

3. App
   - \(\Gamma_3 = [f: t_f]\)
   - \(T_3 = \text{t}_3\)
   - \(S_3 = [\text{int->t}_3/t_f]\)
   - \(\Gamma = [f: t_f]\)
   - \(T = t_f\)
   - \(S = []\)

4. Abs
   - \(\Gamma_4 = S_2(\Gamma_1) = []\)
   - \(T_4 = t_x \rightarrow t_x\)
   - \(S_4 = []\)
   - \(\Gamma = [x: t_x]\)
   - \(T = t_x\)
   - \(S = []\)

Steps at 1, finally:
1. unify( (\text{int->t}_3) \rightarrow \text{t}_3, (\text{t}_x \rightarrow \text{t}_x) \rightarrow \text{t}_1 )
   returns \(S = [\text{int/t}_1] [\text{int/t}_3] [\text{int/t}_x] [\text{int->t}_3/t_f]\)
2. \(S_1 = S \ S_4\ S_2 = S \ S_2 = S [\text{int->t}_3/t_f]\)
3. \(T_1 = S(t_1) = \text{int}\)

from \(\text{Unify}(t_f, \text{int->t}_3)\)
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})
```

Program Analysis CSCI 4450/6450, A Milanova (from MIT 2015 Program Analysis OCW)
def W(\(\Gamma\), E) = case E of

  c    ->  (\([], \text{TypeOf}(c)\))

  x    ->  if (x NOT in \text{Dom}(\(\Gamma\))) then fail
                     else let \(T_E = \Gamma(x)\);
                          in (\([], T_E\))

  \(\lambda x. E_1\) -> let \((S_{E_1}, T_{E_1}) = W(\(\Gamma\)+\{x:t\}, E_1)\)
                                 in \((S_{E_1}, S_{E_1}(t)\rightarrow T_{E_1})\)

  E_1 E_2 -> let \((S_{E_1}, T_{E_1}) = W(\(\Gamma\), E_1)\)
                           \((S_{E_2}, T_{E_2}) = W(S_{E_1}(\(\Gamma\)), E_2)\)
                           \(S = \text{Unify}(S_{E_2}(T_{E_1}), T_{E_2}\rightarrow t)\)
                          in \((S S_{E_2} S_{E_1}, S(t)) \text{// } S S_{E_2} S_{E_1} \text{ composes substitutions}\)

let x = E_1 in E_2 -> let \((S_{E_1}, T_{E_1}) = W(\(\Gamma\)+\{x:t\}, E_1)\)
                          \(S = \text{Unify}(S_{E_1}(t), T_{E_1})\)
                          \((S_{E_2}, T_{E_2}) = W(S S_{E_1}(\(\Gamma\))+\{x:T_{E_1}\}, E_2)\)
                          in \((S_{E_2} S S_{E_1}, T_{E_2})\)
Outline

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

- Parametric polymorphism

- Hindley Milner type inference. Algorithm W
A sound type system rejects some programs that don’t get stuck.

Canonical example

```latex
let f = \lambda x. x
in
  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)
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \mid \forall T.\tau \]
  \[ E ::= x \mid \lambda x:\tau. E \mid E_1 E_2 \mid \Lambda T. E \mid E [\tau] \]
  Can instantiate with polymorphic type!

- Very powerful
  - Can type self application \( \lambda x. x x \)
  - Still cannot type \texttt{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

\[
\tau ::= b \mid \tau_1 \to \tau_2 \mid T
\]

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

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

- 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
  - Strategy 2: on-the-fly typing

- Parametric polymorphism

- Hindley Milner type inference. Algorithm W
let \( f = \lambda x. x \)

in

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

- Constraints

\[
\begin{align*}
    t_f &= t_1 \rightarrow t_1 \\
    t_f &= \text{bool} \rightarrow t_2 \quad // \text{at call } (f \text{ true}) \\
    t_f &= \text{int} \rightarrow t_3 \quad // \text{at call } (f \text{ 1})
\end{align*}
\]

- Doesn’t unify!
Solution:

Generalize the type variable in type of \( f \)

\[
f_t : t_1 \to t_1 \text{ becomes } t_f : \forall T . T \to T
\]

Different uses of generalized type variables are instantiated differently

- E.g., \((f \ true)\) instantiates \( t_f \) into \( \text{bool} \to \text{bool} \)
- E.g., \((f \ 1)\) instantiates \( t_f \) into \( \text{int} \to \text{int} \)

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):
  - \( \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t \)
  - E.g., \text{int}, \text{bool}, \text{int} \rightarrow \text{bool}, \text{t}_1 \rightarrow \text{int}, \text{t}_1 \rightarrow \text{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

\[ \text{Gamma ::= Identifiers} \rightarrow \text{Type schemes} \]

Program Analysis CSCI 4450/6450, A Milanova (from MIT’s 2015 Program Analysis OCW)
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 $\sigma$
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: $t_1 \to t_1$
- Then we’ll generalize that type, $\text{Gen}([], t_1 \to t_1)$: $\forall t_1.t_1 \to t_1$
- 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., $[u_2/t_1] (t_1 \to t_1)$ where $u_2$ is fresh type variable at (f 1)
Generalization, Examples

- \( \lambda f : t_f \cdot \lambda x : t_x \cdot \text{let } g = f \text{ in } g \ x \)
  - 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 \( (\lambda f : t_f \cdot \lambda x : t_x \cdot \text{let } g = f \text{ in } g \ x) \ (\lambda y. y + 1) \text{ true} \) will type-check (unsound!!!)
- DO NOT generalize variables that are mentioned in type environment \( \Gamma \)!!
Hindley Milner Typing Rules

\[
\begin{align*}
\Gamma; x : \tau & \vdash E_1 : \tau & \Gamma; x : \text{Gen}(\Gamma, \tau) & \vdash E_2 : \tau' \\
\Gamma & \vdash \text{let } x = E_1 \text{ in } E_2 : \tau'
\end{align*}
\]

(Let)

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

\[
\begin{align*}
x: \sigma \in \Gamma & \quad \tau < \sigma \\
\Gamma & \vdash x : \tau
\end{align*}
\]

(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, \(c\), \(\text{App}\), \(\text{Abs}\) are as in \(F_1\).)
Hindley Milner Type Inference, Rough Sketch

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 $\forall t₁. t₁ \rightarrow t₁$ so id is instantiated to $u_k \rightarrow u_k$ at (id 1)
Hindley Milner Type Inference

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

let f = \(\lambda x.x\) 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
   \(f \rightarrow \text{true}\)
   \(f \rightarrow 1\)

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

\[ \lambda x. \text{let } f = \lambda y. x \text{ in } (f \text{ true}, f \ 1) \]
Strategy 2: Algorithm W

```python
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_1,...,t_n.τ  -> ( [], [u_1/t_1...u_n/t_n] τ )
                          |                   -> ( [], 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})

// ...
// continues on next slide!
```

u_1 to u_n are fresh type vars generated at instantiation of polymorphic type
Strategy 2: Algorithm W

```python
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 Sₑ₁, Tₑ₂)
```

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

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

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

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

let $x = (\lambda z. z)$

in

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