Simple Type Inference, Polymorphism, Hindley Milner Type Inference

Outline

- Simple type inference
  - Equality constraints
  - Unification
  - Strategy 1: Constraint-based typing
  - Strategy 2: On-the-fly typing: Algorithm W, almost
- Parametric polymorphism
- Hindley Milner type inference. Algorithm W

Type Inference, Strategy 2

- Strategy 1 collects all constraints, then solves them offline
- Strategy 2 solves constraints on the fly
  - Builds the substitution map incrementally
  - Key reason: infers types as parser parses program!

Example: \((\lambda x. f) (\lambda x. x)\)

1. unify \((\text{int} \to \text{int})\) \(\to\) \(\text{int}\)
2. \(\text{Abs} \gamma\) \(\text{Abs} \gamma\)
3. \(\text{App} \lambda x. t\) \(\text{Var} x\)
4. \(\text{Var} f\) \(\text{Const} S\) \(\text{Var} x\)

Add a New Attribute, Substitution Map \(S\)

<table>
<thead>
<tr>
<th>Grammar rule</th>
<th>Attribute rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E ::= x)</td>
<td>(T_E = \Gamma_E(x)) (S_E = {})</td>
</tr>
<tr>
<td>(E ::= c)</td>
<td>(T_E = \text{int}) (S_E = {})</td>
</tr>
<tr>
<td>(E ::= \lambda x.E_i)</td>
<td>(\Gamma_{E_i} = \Gamma_{E_i}) (\Gamma_E = S_{E_i}(\Gamma_{E_i}))</td>
</tr>
<tr>
<td>(E ::= E_1 E_2)</td>
<td>(T_E = S_{E_1}(T_{E_1}) \to T_{E_2} \to T_{E_2} = S_{E_2} = S_{E_2} S_{E_1})</td>
</tr>
</tbody>
</table>

Announcements

- HW6 up
- Quiz 5
- Paper presentation guidelines coming up
- Papers coming up
- HW6 up tomorrow

Example term from MIT 2015 Program Analysis OCW
**Example:** \( \lambda f. \lambda x. (f \circ x) \)

**The let Construct**

- **Typing rule**
  \[
  \Gamma |- E_1 : \sigma \\
  \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau
  \]

- **Attribute grammar rule**
  \[
  E ::= \text{let } x = E_1 \text{ in } E_2 \\
  \Gamma |- E_1 : \sigma \\
  \Gamma |- E_2 : \tau
  \]

  - \( \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau \)
  - \( \Gamma |- E_1 : \sigma \)
  - \( \Gamma |- E_2 : \tau \)

**The letrec Construct**

- **letrec** \( x = E_1 \text{ in } E_2 \)
  
**Attribute grammar rule**

- **letrec** \( x = E_1 \text{ in } E_2 \)
  
  - \( \Gamma |- E_1 : \tau \)
  - \( \Gamma |- E_2 : \tau \)
  - \( \Gamma |- \text{letrec } x = E_1 \text{ in } E_2 : \tau \)

**let/letrec Examples**

- **letrec** \( x = E_1 \text{ in } E_2 \)
  
**Attribute grammar rule**

- **letrec** \( x = E_1 \text{ in } E_2 \)
  
  - \( \Gamma |- E_1 : \tau \)
  - \( \Gamma |- E_2 : \tau \)
  - \( \Gamma |- \text{letrec } x = E_1 \text{ in } E_2 : \tau \)

**let/letrec Examples**

- **letrec** \( x = E_1 \text{ in } E_2 \)
  
**Attribute grammar rule**

- **letrec** \( x = E_1 \text{ in } E_2 \)
  
  - \( \Gamma |- E_1 : \tau \)
  - \( \Gamma |- E_2 : \tau \)
  - \( \Gamma |- \text{letrec } x = E_1 \text{ in } E_2 : \tau \)

**Haskell**

- **let** \( x = E_1 \text{ in } E_2 \)
  
**Attribute grammar rule**

- **let** \( x = E_1 \text{ in } E_2 \)
  
  - \( \Gamma |- E_1 : \tau \)
  - \( \Gamma |- E_2 : \tau \)
  - \( \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau \)

**Haskell**

- **let** \( x = E_1 \text{ in } E_2 \)
  
**Attribute grammar rule**

- **let** \( x = E_1 \text{ in } E_2 \)
  
  - \( \Gamma |- E_1 : \tau \)
  - \( \Gamma |- E_2 : \tau \)
  - \( \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau \)

**Haskell**

- **let** \( x = E_1 \text{ in } E_2 \)
  
**Attribute grammar rule**

- **let** \( x = E_1 \text{ in } E_2 \)
  
  - \( \Gamma |- E_1 : \tau \)
  - \( \Gamma |- E_2 : \tau \)
  - \( \Gamma |- \text{let } x = E_1 \text{ in } E_2 : \tau \)
Algorithm W, Almost There!

\[
\text{def } W(\Gamma, E) = \text{case } E \text{ of }
\]
\[
c \rightarrow (\text{TypeOf}(c))
\]
\[
x \rightarrow \begin{cases} 
\text{if } (x \text{ NOT in Domain}(\Gamma)) \text{ then fail} \\
\text{else let } T_x = \Gamma(x) ; \\
in (\Gamma, T_x) \\
\end{cases}
\]
\[
\lambda x.E_1 \rightarrow \begin{cases} 
\text{let } (S_1, T_{x1}) = W(\Gamma, E_1) \\
(S_1, T_{x1}) = W(S_1, T_{1}) \\
S = \text{Unify}(S_1, T_{x1}) \\
in (S, S_1) \text{ if } S_1 \text{ is } S11 
\end{cases}
\]
\[
\text{let } x = E_1 \text{ in } E_2 \rightarrow \begin{cases} 
\text{let } (S_x, T_{x2}) = W(\Gamma, E_2) \\
(S_{x}, T_{x2}) = W(S_{x}, T_{2}) \\
S = \text{Unify}(S_{x}, T_{x2}) \\
in (S, S_{x}) = S_{x2} 
\end{cases}
\]
\[
\lambda x. E \rightarrow \begin{cases} 
\text{let } (S_x, T_{x1}) = W(\Gamma, E) \\
(S_{x}, T_{x1}) = W(S_{x}, T_{1}) \\
S = \text{Unify}(S_{x}, T_{x1}) \\
in (S, S_{x}) = S_{x2} 
\end{cases}
\]
\[
s1 = W(\Gamma, E_1) \\
s2 = W(\Gamma, E_2) \\
s = g(\Gamma, s1, \Gamma, s2)
\]

Algorithm W, Almost There! (merges let and letrec)

Motivating Example

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

- Canonical example:
  
  let f = \lambda x. x

  in

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

  Term does not get “stuck”

  Term is not typable in the simply typed lambda calculus. But it is typable in Hindley Milner

Explicit Parametric Polymorphism

Java generics

C++ templates

Formalization in the Lambda calculus

\[
\Gamma, \top \vdash E : \tau \quad \text{(Tab)}
\]
\[
\Gamma \vdash E : \forall \tau. \tau' \quad \text{(TApp)}
\]
The Polymorphic Lambda Calculus (System F)

- Adds two rules to System F₁
  - Dynamic semantics
    \[ E₁ \rightarrow E₂ \]
    \[ (\Lambda T. E) [r] \rightarrow E_2 [r] \]
  - E.g., \( (\Lambda T. \lambda x:T. x \, [\text{int}]) \, 1 \rightarrow (\lambda x:\text{int}. x) \, 1 \rightarrow 1 \)

- Static semantics
  \[ \Gamma, T \vdash E : \tau \]

Different Styles of Polymorphism

- Impredicative polymorphism (System F)
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \mid \forall T. \tau \]
  \[ E ::= x \mid \lambda x: \tau. E \mid E_1, E_2 \mid AT.E \mid E [r] \]
  - Can instantiate with polymorphic type!
  - Very powerful
  - Although, still cannot type \textbf{fix}!
  - Type inference is undecidable!

Different Styles of Polymorphism

- Predicative polymorphism
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall T. \sigma \mid \sigma_1 \rightarrow \sigma_2 \]
  \[ E ::= x \mid \lambda x: \tau. E \mid E_1, E_2 \mid AT.E \mid E [r] \]
  - Still very powerful
  - But cannot instantiate with a polymorphic type
  - Type inference is still undecidable!

Different Styles of Polymorphism

- Prenex polymorphism
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall T. \sigma \]
  \[ E ::= x \mid \lambda x: \tau. E \mid E_1, E_2 \mid AT.E \mid E [r] \]
  - 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
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall T. \sigma \]
  \[ E ::= x \mid \lambda x: \tau. E \mid E_1, E_2 \mid AT.E \mid E [\tau] \mid \text{let } x = E_1 \text{ in } E_2 \]
  - Like \( (\lambda x: \tau. E_2) \, E_1 \) but \( x \) can be polymorphic!
  - Good engineering compromise
    - Enhance expressiveness
    - Preserve decidability
  - This is the Hindley Milner type system

Outline

- Simple type inference
  - Equality constraints
  - Unification
    - Simple type inference on-the-fly
    - Algorithm W, almost
- Parametric polymorphism
- Hindley Milner type inference
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 true)
\( t_f = \text{int} \rightarrow t_3 \) // at call (f 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):
- \( \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t \)
- \( t \) is a type variable
- E.g., \( \text{int}, \text{bool}, \text{int} \rightarrow \text{bool}, t_1 \rightarrow t_2, t_1 = t_2, \text{etc.} \)

Type schemes (aka polymorphic types):
- \( \sigma ::= \tau \mid \forall \tau_1 S \sigma \)
- E.g., \( \forall \tau_1. \forall \tau_2. (\text{int} \rightarrow t_1) \rightarrow t_2 \rightarrow t_3 \)
- \( \sigma \) is a “free” type variable as it isn’t bound under \( \forall \)
- Note: all quantifiers appear in the beginning, \( \tau \) cannot contain schemes

Gammas ::= Identifiers \( \rightarrow \) Type schemes

Instantiations

Type scheme \( \sigma = \forall \tau_1 \ldots \tau_n \tau \) can be instantiated into a type \( \tau' \) by substituting types for the bound variables (BV) under the universal quantifier \( \forall \)

- \( \tau' = S \tau \quad S \) is a substitution s.t. \( \text{Domain}(S) \subseteq \text{BV}(\sigma) \)
- \( \tau' \) is said to be an instance of \( \tau \ (\sigma > \tau') \)
- \( \tau \) is said to be a generic instance when \( S \) maps some type variables to new type variables

E.g., \( \sigma = \forall \tau_1 t_1 \rightarrow t_2 \)
\[ [t_2 / t_1] \Rightarrow t_1 \rightarrow t_2 = \text{a generic instance of } \sigma \]
\[ [\text{int} / t_1] \Rightarrow t_1 \rightarrow t_2 = \text{int} \rightarrow t_2 = \text{a non-generic instance of } \sigma \]

Generalization (aka Closing)

We can generalize a type \( \tau \) as follows
\[ \text{Gen}(\Gamma, \tau) = \forall \tau_1 \ldots \tau_n \tau \]
where \( \{ \tau_1, \ldots, \tau_n \} = \text{FV}(\tau) – \text{FV}(\Gamma) \)

- Generalization introduces polymorphism
- Quantify type variables that are free in \( \tau \) but are not free in the type environment \( \Gamma \)
  - E.g., \( \text{Gen}[[t_1], t_1 \rightarrow t_2] \) yields \( \forall \tau_1 t_1 \rightarrow t_2 \)
  - E.g., \( \text{Gen}[[x, t_1], t_1 \rightarrow t_2] \) yields \( \forall \tau_1 t_1 \rightarrow t_2 \)
Generalization, Examples

let \( f = \lambda x. x \) in \((f \text{ true}) \) then \((f \text{ } 1)\) 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
  \( \text{if} (f \text{ true}) \) then \((f \text{ } 1)\) else \(1\)
- E.g., \([t_0 t_1] \) \((t_1 \rightarrow t_1)\) where \(t_0\) is fresh type variable at \((f \text{ } 1)\)

Extend Strategy 2 (Algorithm W)

Two ways:

1. let \( f = \lambda x. x \) in \((f \text{ true}) \) then \((f \text{ } 1)\) else \(1\)
2. \( \text{Gen}([], t_1 \rightarrow t_1) \) yields?
3. Why can’t we generalize \( t_1 \)?
4. Suppose we can generalize to \( \forall t_1 \)
   - Then \( \forall t_1 = t_1 \) will instantiate at \( g \) to some fresh \( t_2 \)
   - Then \( t_0 \) becomes \( t_0 \rightarrow t_0 \) thus losing the important connection between \( t_0 \) and \( t_1 \)
   - Thus \((f:t_0, \lambda x.t_0, \text{let } g = f \text{ in } g \ x) \) \((\forall y.y+1) \text{ true} \) will type-check (unsound!!)
5. DO NOT generalize variables that are mentioned in type environment \( \Gamma \)!

Hindley Milner Typing Rules

\[
\Gamma \vdash x : \tau \quad \text{E}\quad \Gamma ; x : \text{Gen}(\Gamma, \tau) \vdash E_2 : \tau'
\]

(Let)

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

\[
\frac{x : \sigma \subseteq \Gamma \quad \tau \sigma}{\Gamma \vdash x : \tau}
\]

(Var)

- \( x \) in \( E_2 \) of \( \Gamma \) \( x \) considered type \( \tau \) if it’s 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, Sketch

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

1. Calculate type \( T_{E_1} \) for \( E_1 \) in \( \Gamma ; x : t_1 \) 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 \( \forall y.t_1 \rightarrow t_1 \), so \( \text{id} \) is instantiated to \( t_1 \rightarrow t_0 \) at \((\text{id } 1)\)

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

1. let \( \Gamma = [] \)
2. \( t_2 = \lambda x. t_1 \)
3. \( t_1 = t_1 \rightarrow t_2 \)
4. \( t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow \text{int } t_5 \)

Solve (!) constraint \( t_1 = t_1 \rightarrow t_2 \) \((t_0 \rightarrow t_5)\)
Next, generalize \( t_0 \) \((t_0 \rightarrow t_5)\)

\( t_1 \) and \( t_2 \) are fresh type vars generated at instantiation of polymorphic type.
Example

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

Strategy 2: Algorithm W

```
def W(\Gamma, E) = case E of
  c -> ([], TypeOf(c))
  x -> if (x \text{ NOT in Domain(}\Gamma)) \text{ then fail }
      else let \Gamma_x = \Gamma(x)
        \text{ in case } \Gamma_x \text{ of }
        \forall t_1 \ldots t_n -> ([\{u_1/t_1, \ldots, u_n/t_n\} \Gamma], \)
        \text{ in case } \Gamma_2 \text{ of }
        \lambda x. E_1 -> let \Gamma_{x_1} = \Gamma(x) + \{x: t_1\}
        \text{ in } (\Gamma_{x_1}, E_1)
        \text{ in } (\Gamma_2, \Gamma_{x_2}, \Gamma_{x_3} = \Gamma(x) + \{x: t_1\})

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

Example

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

Strategy 2 Example

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

1. let \Gamma = [] \text{ T_s = int S_s = [ ]}
2. Abs \text{ T_s = } t_x \text{ S_s = [ ]}
3. if-then-else \text{ T_s = int S_s = [ ]}
4. App \text{ f = } \lambda x.\text{\textbf{in}} \text{ if (f true) then (f 1) else 1}
5. App \text{ f = true}

\text{E} \text{\textit{Escapes constraint, types 2. Abs immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
\text{\textit{T_s = t_x}} \text{\textit{immediately:}}
```

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. Inferred types are verifiable
- Complexity of Algorithm W
  - PSPACE-Hard
  - Because of nested let blocks

Hindley Milner Limitations

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

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