Type Inference and the Hindley-Milner Type System

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.
With slides from Arvind

September 29, 2011
Type Inference

- Consider the following expression
  - \((\lambda f:\text{int}\to\text{int}. f\ 5)\ (\lambda x:\text{int}. x + 1)\)
- Is it well typed in \(F_1\)?

\[
\frac{x: \tau \in \Gamma}{\Gamma \vdash x : \tau}
\]

\[
\frac{\Gamma, x: \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x: \tau_1\ e) : \tau_1 \to \tau_2}
\]

\[
\frac{\Gamma \vdash e_1: \tau' \to \tau \quad \Gamma \vdash e_2: \tau'}{\Gamma \vdash e_1 e_2: \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
\]
There wasn’t a single point in the derivation where we had to look at the type labels in order to know what rule to apply!
- We could have written the derivation without the labels.

The labels helped us determine the actual types for all the $\tau$s in the typing rules.
- We could have figured these out even without the labels.
- This is the key idea behind type inference!
Type Inference Strategy

1. Use the typing rules to define constraints on the possible types of expressions

2. Solve the resulting constraint system
The language of Equality Constraints

- Consider the following Language of Constraints
  \[ C ::= \tau_1 = \tau_2 | C \land C | \exists \tau. C \]

- Constraints in this language have a lot of good properties
  - Nice and compositional
  - Linear time solution algorithm
Building Constraints from Typing Rules

• Notation
  \([\text{Judgment}] = \text{Constraints}\)
  - The constraints on the right ensure that the judgment on the left holds
  - This mapping is defined recursively.

• Base cases
  \([\Gamma \vdash x: \tau] = \Gamma(x) = \tau\)
  \([\Gamma \vdash N: \tau] = \text{int} = \tau\)

• Inductive Cases
  \([\Gamma \vdash e_1 e_2: \tau] = \exists a_1 a_2. (\ [\Gamma \vdash e_1: a \rightarrow \tau] \wedge \ [\Gamma \vdash e_2: a] )\)
  \([\Gamma \vdash \lambda x. e: \tau] = \exists a_1 a_2. (\ [\Gamma; x: a_1 \vdash e: a_2] \wedge \tau = a_1 \rightarrow a_2 )\)
  \([\Gamma \vdash e_1 + e_2: \tau] = (\ [\Gamma \vdash e_1: \text{int}] \wedge \ [\Gamma \vdash e_2: \text{int}] )\)
Back to our example

$$(\lambda f: \text{int}\to \text{int}. f\ 5)\ (\lambda x: \text{int}. x + 1)$$

$$[[\Gamma \vdash x : \tau]] = \Gamma(x) = \tau$$

$$[[\Gamma \vdash N : \tau]] = \text{int} = \tau$$

$$[[\Gamma \vdash e_1 e_2 : \tau]] = \exists a ([[\Gamma \vdash e_1 : a \to \tau]] \land [[\Gamma \vdash e_2 : a]])$$

$$[[\Gamma \vdash \lambda x. e : \tau]] = \exists a_1 a_2. ([[[\Gamma ; x : a_1 \vdash e : a_2]]] \land \tau = a_1 \to a_2)$$

$$[[\Gamma \vdash e_1 + e_2 : \tau]] = ([[[\Gamma \vdash e_1 : \text{int}]]] \land [[[\Gamma \vdash e_2 : \text{int}]])$$
Simple Inference Algorithm (cont-1)

\[
\text{Def } W(TE, e) = \\
\text{Case } e \text{ of } \\
\begin{align*}
\text{c} & = (\{\}, \text{Typeof}(c)) \\
\text{x} & = \text{if } (x \notin \text{Dom}(TE)) \text{ then Fail} \\
& \quad \text{else let } \tau = \text{TE}(x); \\
& \quad \text{in } (\{\}, \tau)
\end{align*}
\]

\[
[\Gamma \vdash \lambda x.e : \tau] = \exists a_1a_2. ( [\Gamma; x : a_1 \vdash e : a_2] \land \tau = a_1 \rightarrow a_2 )
\]
\[
\lambda x.e \quad = \text{let } (S_1, \tau_1) = W(TE + \{ x : u \}, e) \\
\quad \text{in } (S_1, S_1(u) \rightarrow \tau_1)
\]

\[
[\Gamma \vdash e_1e_2 : \tau] = \exists a ( [\Gamma \vdash e_1 : a \rightarrow \tau] \land [\Gamma \vdash e_2 : a] )
\]
\[
(e_1 \; e_2) \quad = \text{let } (S_1, \tau_1) = W(TE, e_1); \\
\quad (S_2, \tau_2) = W(S_1(TE), e_2); \\
\quad S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \rightarrow u); \\
\quad \text{in } (S_3 S_2 S_1, S_3(u))
\]

u’s represent new type variables
Simple Inference Algorithm (cont-1)

Def $W(TE, e) =$

Case $e$ of

$c =$ ({}, Typeof(c))

$x =$ \[ if \ (x \not\in \text{Dom}(TE)) \ then \ \text{Fail} \]

\hspace{1cm} else \ let \ \tau = TE(x);

\hspace{1cm} in \ ({}, \tau) $

$\lambda x. e =$ let $(S_1, \tau_1) = W(TE + \{x : u\}, e)$

\hspace{1cm} in $(S_1, S_1(u) \rightarrow \tau_1)$

$(e_1 \ e_2) =$ let $(S_1, \tau_1) = W(TE, e_1)$

\hspace{1cm} $(S_2, \tau_2) = W(S_1(TE), e_2)$;

\hspace{1cm} $S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \rightarrow u)$;

\hspace{1cm} in $(S_3 \ S_2 \ S_1, S_3(u))$

let $x = e_1$ in $e_2$

\hspace{1cm} = let $(S_1, \tau_1) = W(TE + \{x : u\}, e_1)$;

\hspace{1cm} $S_2 = \text{Unify}(S_1(u), \tau_1)$;

\hspace{1cm} $(S_3, \tau_2) = W(S_2 \ S_1(TE) + \{x : \tau_1\}, e_2)$;

\hspace{1cm} in $(S_3 \ S_2 \ S_1, \tau_2)$

u’s represent new type variables
Polymorphism
Some observations

- A type system restricts the class of programs that are considered “legal”
- It is possible a term in the untyped λ–calculus may be reducible to a value but may not be typeable in a particular type system

\[
\begin{array}{l}
\text{let} \\
\quad \text{id} = \lambda x. x \\
\text{in} \\
\quad \ldots (\text{id} \ True) \ldots (\text{id} \ 1) \ldots
\end{array}
\]

This term is not typeable in the simple type system we have discussed so far. However, it is typeable in the Hindley-Milner system
Explicit polymorphism

• You’ve seen this before

```
public interface List<E>{
    void add(E x);
    E get();
}
```

List<String> ls = ...
ls.add("Hello");
String hello = ls.get(0);

• How do we formalize this?

\[
\Gamma \vdash e : \tau \\
\Gamma \vdash \Lambda t. e : \forall t. \tau
\]

• Example

\[
id = \Lambda T. \lambda x : T. x
\]

\[
id[int] 5
\]
Different Styles of Polymorphism

- Impredicative Polymorphism
  \[\tau ::= b \mid \tau_1 \to \tau_2 \mid T \mid \forall T. \tau\]
  \[e ::= x \mid \lambda x: \tau. e \mid e_1 e_2 \mid \Lambda T. e \mid e[\tau]\]

- Very powerful
  - Although you still can’t express recursion

- Type inference is undecidable!
Different Styles of Polymorphism

- Predicative Polymorphism
  \[ \tau ::= b \mid \tau_1 \to \tau_2 \mid T \]
  \[ \sigma ::= \tau \mid \forall T. \sigma \mid \sigma_1 \to \sigma_2 \]
  \[ e ::= x \mid \lambda x: \sigma. \ e \mid e_1 e_2 \mid \Lambda T. \ e \mid e[\tau] \]

- Still very powerful
  - But you can no longer instantiate with a polymorphic type

- Type inference is still undecidable!
Different Styles of Polymorphism

• Prenex Predicative 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 we have decidable type inference
• But polymorphism is now very limited
  – We can’t pass polymorphic functions as arguments!!
  – \((\lambda s: \forall T. \tau \ldots s[\text{int}]x \ldots s[\text{bool}]x)(\Lambda T. \text{code for sort})\)
Let polymorphism

• Introduce let x = e1 in e2
  – Just like saying \((\lambda x.e2)e1\)
  – Except x can be polymorphic

• Good engineering compromise
  – Enhance expressiveness
  – Preserve decidability
Type inference with polymorphism
Polymorphic Types

```latex
let
  id = \lambda x. x
in
... (id True) ... (id 1) ...
```

Constraints:

```latex
id :: t_1 --> t_1
id :: Int --> t_2
id :: Bool --> t_3
```

Does not unify!!

Solution: Generalize the type variable

```latex
id :: \forall t_1. t_1 --> t_1
```

Different uses of a generalized type variable may be instantiated differently

- \( id_2 : \text{Bool} --> \text{Bool} \)
- \( id_1 : \text{Int} --> \text{Int} \)

When can we generalize?
A mini Language
to study Hindley-Milner Types

Expressions

\[ E ::= \text{\texttt{c}} \quad \text{constant} \]
\[ \mid x \quad \text{variable} \]
\[ \mid \lambda x. E \quad \text{abstraction} \]
\[ \mid (E_1 \ E_2) \quad \text{application} \]
\[ \mid \text{let } x = E_1 \text{ in } E_2 \quad \text{let-block} \]

- There are no types in the syntax of the language!

- The type of each subexpression is derived by the Hindley-Milner type inference algorithm.
A Formal Type System

Types
\[ \begin{align*}
\tau &::= \iota \\
& \mid t \\
& \mid \tau_1 \rightarrow \tau_2
\end{align*} \]

Type Schemes
\[ \begin{align*}
\sigma &::= \tau \\
& \mid \forall t. \sigma
\end{align*} \]

Type Environments
\[ \text{TE ::= Identifiers} \rightarrow \text{Type Schemes} \]

Note, all the \( \forall \)'s occur in the beginning of a type scheme, i.e., a type \( \tau \) cannot contain a type scheme \( \sigma \)
Instantiations

\[ \sigma = \forall t_1 \ldots t_n. \tau \]

- Type scheme \( \sigma \) can be \textit{instantiated} into a type \( \tau' \) by \textit{substituting types for the bound variables} of \( \sigma \), i.e.,

\[ \tau' = S \tau \quad \text{for some } S \text{ such that } \text{Dom}(S) \subseteq \text{BV}(\sigma) \]

- \( \tau' \) is said to be an \textit{instance} of \( \sigma \) \( (\sigma > \tau') \)

- \( \tau' \) is said to be a \textit{generic instance} of \( \sigma \) when \( S \) maps variables to new variables.

Example:

\[ \sigma = \forall t_1. t_1 \rightarrow t_2 \]

\( t_3 \rightarrow t_2 \) is a generic instance of \( \sigma \)

\( \text{Int} \rightarrow t_2 \) is a non generic instance of \( \sigma \)
Generalization *aka Closing*

\[
\text{Gen}(\text{TE}, \tau) = \forall t_1 \ldots t_n. \tau \\
\text{where } \{ t_1 \ldots t_n \} = \text{FV}(\tau) - \text{FV}(\text{TE})
\]

- *Generalization* introduces polymorphism
- Quantify type variables that are free in \( \tau \) but not *free* in the type environment (TE)
- Captures the notion of *new* type variables of \( \tau \)
HM Type Inference Rules

**Typing:**
\[
\Gamma \vdash e : \tau
\]

**(App)**
\[
\begin{align*}
\Gamma &\vdash e_1 : \tau \rightarrow \tau' \\
\Gamma &\vdash e_2 : \tau \\
\Gamma &\vdash (e_1 e_2) : \tau'
\end{align*}
\]

**(Abs)**
\[
\begin{align*}
\Gamma + \{x : \tau\} &\vdash e : \tau' \\
\Gamma &\vdash \lambda x. e : \tau \rightarrow \tau'
\end{align*}
\]

**(Var)**
\[
\begin{align*}
(x : \sigma) &\in \Gamma \\
\Gamma &\vdash x : \tau
\end{align*}
\]

**(Const)**
\[
\begin{align*}
\text{typeof(c)} &\geq \tau \\
\Gamma &\vdash c : \tau
\end{align*}
\]

**(Let)**
\[
\begin{align*}
\Gamma + \{x : \tau\} &\vdash e_1 : \tau \\
\Gamma + \{x: \text{Gen}(\Gamma, \tau)\} &\vdash e_2 : \tau'
\end{align*}
\]
\[
\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau'
\]
HM Inference Algorithm

Def \( W(TE, e) \) = Case \( e \) of

\[\begin{align*}
  c &= (\{\}, Typeof(c)) \\
  x &= if (x \notin \text{Dom}(TE)) then \text{Fail} \\
    &\quad else let \quad \forall t_1...t_n.\tau = TE(x); \\
    &\quad \quad in \quad (\{\}, [u_i / t_i] \tau) \\
  \lambda x.e &= let \quad (S_1, \tau_1) = W(TE + \{x : u\}, e); \\
    &\quad in \quad (S_1, S_1(u) -> \tau_1) \\
  (e_1 e_2) &= let \quad (S_1, \tau_1) = W(TE, e_1); \\
    &\quad (S_2, \tau_2) = W(S_1(TE), e_2); \\
    &\quad S_3 = \text{Unify}(S_2(\tau_1), \tau_2 -> u); \\
    &\quad in \quad (S_3 S_2 S_1, S_3(u)) \\
  let \ x = e_1 \ in \ e_2 \\
    &= let \quad (S_1, \tau_1) = W(TE + \{x : u\}, e_1); \\
    &\quad S_2 = \text{Unify}(S_1(u), \tau_1); \\
    &\quad \sigma = \text{Gen}(S_2 S_1(TE), S_2(\tau_1)); \\
    &\quad (S_3, \tau_2) = W(S_2 S_1(TE) + \{x : \sigma\}, e_2); \\
    &\quad in \quad (S_3 S_2 S_1, \tau_2)
\end{align*}\]
Hindley-Milner: Example

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

\[
W(\emptyset, A) = (\ [ \ ] , \ u_1 -> (u_1,u_1) )
\]

\[
W(\{x : u_1\}, B) = (\ [ \ ] , (u_1,u_1) )
\]

\[
W(\{x : u_1, f : u_2\}, \lambda y. x) = (\ [ \ ] , \ u_3 -> u_1 )
\]

\[
W(\{x : u_1, f : u_2, y : u_3\}, x) = (\ [ \ ] , \ u_1 )
\]

\[
\text{Unify}(u_2, u_3 -> u_1) = [ (u_3 -> u_1) / u_2 ]
\]

\[
\text{Gen}(\{x : u_1\}, u_3 -> u_1) = \forall u_3. u_3 -> u_1
\]

\[
\text{TE} = \{x : u_1, f : \forall u_3. u_3 -> u_1\}
\]

\[
W(\text{TE}, (f \ 1)) = (\ [ \ ] , \ u_1 )
\]

\[
W(\text{TE}, f) = (\ [ \ ] , \ u_4 -> u_1 )
\]

\[
W(\text{TE}, 1) = (\ [ \ ] , \ \text{Int} )
\]

\[
\text{Unify}(u_4 -> u_1, \text{Int} -> u_5) = [ \ \text{Int} / u_4 , u_1 / u_5 ]
\]

...
Important Observations

• Do not generalize over type variables used elsewhere

• Let is the only way of defining polymorphic constructs

• Generalize the types of let-bound identifiers only after processing their definitions
Properties of HM Type Inference

• It is sound with respect to the type system. An inferred type is verifiable.

• It generates most general types of expressions. Any verifiable type is inferred.

• Complexity
  PSPACE-Hard
  Nested \textit{let} blocks
Extensions

• Type Declarations
  Sanity check; can relax restrictions

• Incremental Type checking
  The whole program is not given at the same time, sound inferencing when types of some functions are not known

• Typing references to mutable objects
  Hindley-Milner system is unsound for a language with refs (mutable locations)

• Overloading Resolution
HM Limitations: 
λ-bound vs Let-bound Variables

Only let-bound identifiers can be instantiated differently.

```
let
  twice f x = f (f x)
in
  twice twice succ 4
```

versus

```
let
  twice f x = f (f x)
in
  foo g = (g g succ) 4

foo twice
```

foo is not type correct!

Generic vs. Non-generic type variables
Puzzle: Another set of Inference rules

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(Gen)</strong></td>
<td>$\text{TE} \vdash e : \tau$ \hspace{1cm} $t \notin \text{FV(TE)}$</td>
</tr>
<tr>
<td></td>
<td>$\text{TE} \vdash e : \forall t.\tau$</td>
</tr>
<tr>
<td><strong>(Spec)</strong></td>
<td>$\text{TE} \vdash e : \forall t.\tau$</td>
</tr>
<tr>
<td></td>
<td>$\text{TE} \vdash e : \tau [u/t]$</td>
</tr>
<tr>
<td><strong>(Var)</strong></td>
<td>$(x : \tau) \in \text{TE}$</td>
</tr>
<tr>
<td></td>
<td>$\text{TE} \vdash x : \tau$</td>
</tr>
<tr>
<td><strong>(Let)</strong></td>
<td>$\text{TE+}{x:\tau} \vdash e_1 : \tau$ \hspace{1cm} $\text{TE+}{x:\tau} \vdash e_2 : \tau'$</td>
</tr>
<tr>
<td></td>
<td>$\text{TE} \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau'$</td>
</tr>
</tbody>
</table>

(App) and (Abs) rules remain unchanged.

Sound but no inference algorithm!