Some observations

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

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

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

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

Constraints:

- \( id :: t_1 --> t_1 \)
- \( id :: \text{Int} --> t_2 \)
- \( id :: \text{Bool} --> t_3 \)

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

- \( E ::= c \) constant
- \( x \) variable
- \( \lambda x. E \) abstraction
- \( (E_1 E_2) \) application
- \( \text{let } x = E_1 \text{ in } E_2 \) 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
\[ \tau ::= \iota \mid t \mid \tau_1 \rightarrow \tau_2 \]

Type Schemes
\[ \sigma ::= \tau \mid \forall t. \sigma \]

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

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 instantiated into a type \( \tau' \) by substituting types for the bound variables of \( \sigma \), i.e.,
  \[ \tau' = S \tau \quad \text{for some } S \text{ s.t. } \text{Dom}(S) \subseteq \text{BV}(\sigma) \]
- \( \tau' \) is said to be an instance of \( \sigma \) (\( \sigma > \tau' \))
- \( \tau' \) is said to be a generic instance of \( \sigma \) when \( S \) maps variables to new variables.

Example:
\[ \sigma = \forall t_1. t_1 \rightarrow t_2 \]
\[ t_3 \rightarrow t_2 \text{ is a generic instance of } \sigma \]
\[ \text{Int} \rightarrow t_2 \text{ is a non generic instance of } \sigma \]
Generalization *aka* Closing

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

- *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:**
\[ \text{TE} \vdash e : \tau \]

**App**
\[ \text{TE} \vdash e_1 : \tau \to \tau' \quad \text{TE} \vdash e_2 : \tau \]
\[ \text{TE} \vdash (e_1 \; e_2) : \tau' \]

**Abs**
\[ \text{TE} + \{ x : \tau \} \vdash e : \tau' \]
\[ \text{TE} \vdash \lambda x.e : \tau \to \tau' \]

**Var**
\[ (x : \sigma) \in \text{TE} \]
\[ \sigma \geq \tau \]
\[ \text{TE} \vdash x : \tau \]

**Const**
\[ \text{typeof}(c) \geq \tau \]
\[ \text{TE} \vdash c : \tau \]

**Let**
\[ \text{TE} + \{ x : \tau \} \vdash e_1 : \tau \quad \text{TE} + \{ x : \text{Gen}(\text{TE}, \tau) \} \vdash e_2 : \tau' \]
\[ \text{TE} \vdash (\text{let} \; x = e_1 \; \text{in} \; e_2) : \tau' \]
HM Inference Algorithm

**Def** \( W(\text{TE}, e) = \text{Case } e \text{ of} \)

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

Hindley-Milner: Example

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

\[
\begin{align*}
W(\emptyset, A) & = ([], u_1 \rightarrow (u_1, u_1)) \\
W(\{x : u_1\}, B) & = ([], (u_1, u_1)) \\
W(\{x : u_1, f : u_2\}, \lambda y. x) & = ([], u_3 \rightarrow u_1) \\
W(\{x : u_1, f : u_2, y : u_3\}, x) & = ([], u_1) \\
\text{Unify}(u_2, u_3 \rightarrow u_1) & = [u_3 \rightarrow u_1] / u_2] \\
\text{Gen}(\{x : u_1\}, u_3 \rightarrow u_1) & = \forall u_3, u_3 \rightarrow u_1 \\
\text{TE} & = \{x : u_1, f : \forall u_3, u_3 \rightarrow u_1\} \\
W(\text{TE}, (f 1)) & = ([], u_1) \\
W(\text{TE}, f) & = ([], u_4 \rightarrow u_1) \\
W(\text{TE}, 1) & = ([], \text{Int}) \\
\text{Unify}(u_4 \rightarrow u_1, \text{Int} \rightarrow u_5) & = [\text{Int} / u_4, u_1 / u_5]
\end{align*}
\]
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
  - DEXPTIME-Complete
  - Nested 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:
\(\lambda\)-bound vs Let-bound Variables

Only let-bound identifiers can be instantiated differently.

\[
\text{let}\quad \text{twice } f \ x = f \ (f \ x) \\
\text{in}\quad \text{twice} \ \text{twice} \ \text{succ} \ 4
\]

versus

\[
\text{let}\quad \text{twice } f \ x = f \ (f \ x) \\
\text{in}\quad \text{foo} \ \text{g} = (\text{g} \ \text{g} \ \text{succ}) \ 4
\]

\text{foo} \ \text{is not type correct!}

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Gen)</td>
<td>$\text{TE} \vdash e : \tau \quad \tau \notin \text{FV(TE)}$</td>
</tr>
<tr>
<td>(Spec)</td>
<td>$\text{TE} \vdash e : \forall t.\tau$</td>
</tr>
<tr>
<td>(Var)</td>
<td>$(x : \tau) \in \text{TE}$</td>
</tr>
<tr>
<td>(Let)</td>
<td>$\text{TE} + {x : \tau} \vdash e_1 : \tau$</td>
</tr>
<tr>
<td>(App) and (Abs) rules remain unchanged.</td>
<td></td>
</tr>
</tbody>
</table>

Soundness

- The proposed type system is said to be **sound** if $e : \tau$ then $e$ indeed evaluates to a value in $\tau$.

- To prove soundness, we need to show two properties
  
  - **Preservation**: $\text{TE} \vdash e : \tau$ and $(e \rightarrow e') \Rightarrow \text{TE} \vdash e' : \tau$
  
  - **Progress**: $\text{TE} \vdash e : \tau \Rightarrow$
    - Either $e$ is a value or $\exists e'$ s.t. $(e \rightarrow e')$
Type Preservation: Pure $\lambda$-calculus

• Recall our Typing Rules

$\begin{align*}
\Gamma, x : \tau \vdash x : \tau \\
\Gamma, x : \tau_1 \vdash e : \tau_2 \\
\Gamma, x : \tau_1 \vdash e : \tau_2
\end{align*}$

$\begin{align*}
\Gamma : \tau' \vdash e_1 : \tau' \\
\Gamma : \tau \vdash e_2 : \tau
\end{align*}$

• Evaluation Rules

$\begin{align*}
x \to x \\
\lambda x. e \to \lambda x. e
\end{align*}$

$\begin{align*}
e_1 \to \lambda x. e' \\
e_1[e_2[\alpha(e_2)/x]] \to e_3
\end{align*}$

$\begin{align*}
e_1 e_2 \to e_3
\end{align*}$

• We want to show that

$\text{TE} \vdash e : \tau \text{ and } (e \to e') \Rightarrow \text{TE} \vdash e' : \tau$

Induction on the Structure of the Derivation

$\text{TE} \vdash e : \tau \text{ and } (e \to e') \Rightarrow \text{TE} \vdash e' : \tau$

• Base case:

$\begin{align*}
x \to x \\
\lambda x. e \to \lambda x. e
\end{align*}$

$e = e'$, so property is trivially satisfied

• Inductive case

$\begin{align*}
e_1 \to \lambda x. e' \\
e_1[e_2[\alpha(e_2)/x]] \to e_3
\end{align*}$

$\begin{align*}
e_1 e_2 \to e_3
\end{align*}$
Induction on the Structure of the Derivation

- Inductive case \( e_1 \rightarrow \lambda x. e'_1 e'[a(e_2)/x] \rightarrow e_3 \)

  - Given \( \Gamma \vdash e_1 e_2 : \tau_{\xi_{12}} \) we want to show that \( \Gamma \vdash e_3 : \tau_{\xi_{12}} \)
  - By our typing rule, we have
    \[
    \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau_{\xi_{12}} \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau_{\xi_{12}}}
    \]
  - And by the IH, we have that \( \lambda x. e'_1 : \tau' \rightarrow \tau_{\xi_{12}} \)
  - Which again by the typing rule
    \[
    \frac{\Gamma, x: \tau' \vdash e'_1 : \tau_{\xi_{12}} \quad \Gamma \vdash e_2 : \tau' = \Gamma \vdash e'_1[a(e_2)/x] : \tau_{\xi_{12}}}{\Gamma \vdash (\lambda x: \tau' e'_1) : \tau \rightarrow \tau_{\xi_{12}}}
    \]
  - Now, we need to show that
    \[
    \Gamma, x: \tau' \vdash e'_1 : \tau_{\xi_{12}} \land \Gamma \vdash e_2 : \tau' \Rightarrow \Gamma \vdash e'_1[a(e_2)/x] : \tau_{\xi_{12}}
    \]
  - And from our IH
    \[
    \Gamma \vdash e'_1[a(e_2)/x] : \tau_{\xi_{12}} = \Gamma \vdash e_3 : \tau_{\xi_{12}}
    \]

The Language

- Contexts
  \[
  H ::= o \mid H \ e_1 \mid H + e \mid e + H \mid \text{if } H \text{ then } e_1 \text{ else } e_2
  \]

- Local Reduction Rules
  - \( n_1 + n_2 \rightarrow n_1 + n_2 \)
  - \( \text{if true then } e_1 \text{ else } e_2 \rightarrow e_1 \)
  - \( \text{if false then } e_1 \text{ else } e_2 \rightarrow e_2 \)
  - \((\lambda x: r. e_1) v_2 \rightarrow [v_2/x] e_1\)

- Global Reduction Rules
  - \( H[r] \rightarrow H[e] \text{ iff } r \rightarrow e \)
The proof strategy

- **Progress Theorem**
  
  If $\vdash e : \tau$ and $e$ is not a value, then there is an $e'$ s.t. $e \rightarrow e'$

- **We can prove this through a decomposition lemma**
  
  - If $\vdash e : \tau$ and $e$ is not a value, then there are $H$ and $r$ s.t. $e = H[r]
  
  - This guarantees one step of progress

Proving the Progress Theorem

If $\vdash e : \tau$ and $e$ is not a value, then there is an $e'$ s.t. $e \rightarrow e'$

or equivalently, $e = H[r]$  

- **Proved by induction on the derivation of $\vdash e : \tau$**

- **Base case:**
  - Irreducible values
Proving the Progress Theorem

• Inductive case

\[ \Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau \]
\[ \Gamma \vdash \text{if } e \text{ then } e_t \text{ else } e_f : \tau \]

- by the IH, \( e \) can be irreducible,
  • in which case it must be true or false and the whole thing is a redex
- Or, it can be decomposed into \( H[r] \)
  • in which case if \( H \) then \( e_1 \) else \( e_2 \) is a valid context.