Type Inference and the Hindley-Milner Type System

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

September 29, 2011
Type Inference

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

\[
\begin{array}{c}
\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau} \\
\frac{\Gamma, x: \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x: \tau_1 \ e) : \tau_1 \rightarrow \tau_2} \\
\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}
\end{array}
\]

\[
\begin{array}{c}
\frac{\Gamma \vdash N : \text{int}}{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}} \\
\frac{\Gamma \vdash e_1 + e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
\end{array}
\]
Type Inference

- 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

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

• 2. Solve the resulting constraint system
Deducing Types

1. Assign types to every subexpression
   \[ x :: t_0 \quad f :: t_1 \]
   \[ f \ x :: t_2 \quad f \ (f \ x) :: t_3 \]
   \( \Rightarrow \) twice :: t_1 \rightarrow t_0 \rightarrow t_3

2. Set up the constraints
   \[ t_1 = t_0 \rightarrow t_2 \quad \text{because of } (f \ x) \]
   \[ t_1 = t_2 \rightarrow t_3 \quad \text{because of } f \ (f \ x) \]

3. Resolve the constraints
   \[ t_0 \rightarrow t_2 = t_2 \rightarrow t_3 \]
   \( \Rightarrow \) t_0 = t_2 and t_2 = t_3 \( \Rightarrow \) t_0 = t_2 = t_3
   \( \Rightarrow \) twice :: (t_0 \rightarrow t_0) \rightarrow t_0 \rightarrow t_0
The language of Equality Constraints

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

• 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

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

• Inductive Cases

\([\Gamma \vdash e_1 e_2: \tau] = \exists a_1 a_2 ( [\Gamma \vdash e_1: a \to \tau] \land [\Gamma \vdash e_2: a] )\)

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

\([\Gamma \vdash e_1 + e_2: \tau] = [\Gamma \vdash e_1: \text{int}] \land [\Gamma \vdash e_2: \text{int}] \land \tau = \text{int}\)
Back to our example

$$(\lambda f. \: f \: 5) \: (\lambda x. \: x + 1)$$

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

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

$$[\Gamma \vdash e_1 \: e_2: \tau] = \exists a_1 \: a_2 \: (\exists [\Gamma; x: a_1 \vdash e: a_2] \wedge \tau = a_1 \rightarrow a_2)$$

$$[\Gamma \vdash e_1 + e_2: \tau] = [\Gamma \vdash e_1: \text{int}] \wedge [\Gamma \vdash e_2: \text{int}] \wedge \tau = \text{int}$$
Equality and Unification

• What does it mean for two types $\tau_a$ and $\tau_b$ to be equal?
  - *Structural Equality*
    Suppose $\tau_a = \tau_1 \rightarrow \tau_2$
    $\tau_b = \tau_3 \rightarrow \tau_4$
    Is $\tau_a = \tau_b$?
  
  iff $\tau_1 = \tau_3$ and $\tau_2 = \tau_4$

• Can two types be made equal by choosing appropriate substitutions for their type variables?
  - *Robinson’s unification algorithm*
    Suppose $\tau_a = t_1 \rightarrow \text{Bool}$
    $\tau_b = \text{Int} \rightarrow t_2$
    Are $\tau_a$ and $\tau_b$ unifiable?
    
    if $t_1 = \text{Int}$ and $t_2 = \text{Bool}$

    Suppose $\tau_a = t_1 \rightarrow \text{Bool}$
    $\tau_b = \text{Int} \rightarrow \text{Int}$
    Are $\tau_a$ and $\tau_b$ unifiable?
    No
Simple Type Substitutions
needed to define type unification

A substitution is a map
S : Type Variables → Types
S = [τ₁ / t₁, ..., τₙ / tₙ]
τ' = S τ  τ' is a Substitution Instance of τ

Example:
S = [(t -> Bool) / t₁]
S ( t₁ -> t₁) = ( t -> Bool) -> ( t -> Bool) ?

Substitutions can be composed, i.e., S₂ S₁

Example:
S₁ = [(t -> Bool) / t₁] ; S₂ = [Int / t]
S₂ S₁ ( t₁ -> t₁) = S₂ (( t -> Bool) -> ( t -> Bool))
  = ( Int -> Bool) -> ( Int -> Bool) ?

Types
τ ::= t
    | t
    | τ₁ -> τ₂

base types (Int, Bool ..)
type variables
Function types

September 27, 2011
Unification
An essential subroutine for type inference

Unify($\tau_1$, $\tau_2$) tries to unify $\tau_1$ and $\tau_2$ and returns a substitution if successful

\[
\text{def Unify}(\tau_1, \tau_2) =
\begin{cases}
\text{case } (\tau_1, \tau_2) \text{ of} \\
(\tau_1, t_2) = [\tau_1 / t_2] \text{ provided } t_2 \notin \text{FV}(\tau_1) \\
(t_1, \tau_2) = [\tau_2 / t_1] \text{ provided } t_1 \notin \text{FV}(\tau_2) \\
(t_1, t_2) = \text{if } (\text{eq? } t_1 t_2) \text{ then } [ ] \text{ else } \text{fail}
\end{cases}
\]

\[
(\tau_{11} \rightarrow \tau_{12}, \tau_{21} \rightarrow \tau_{22})
= \begin{cases}
\text{let } S_1 = \text{Unify}(\tau_{11}, \tau_{21}) \\
S_2 = \text{Unify}(S_1(\tau_{12}), S_1(\tau_{22})) \\
in S_2 S_1
\end{cases}
\]

\begin{align*}
\text{otherwise } &= \text{fail}
\end{align*}

Does the order matter? No
Type inference strategy 2

• Like strategy 1, but we solve the constraints as we see them
  – Build the substitution map incrementally
Simple Inference Algorithm

\[ W(TE, e) \text{ returns } (S, \tau) \text{ such that } S (TE) \vdash e : \tau \]

The type environment \( TE \) records the most general type of each identifier while the substitution \( S \) records the changes in the type variables.

\[
\text{Def } W(TE, e) = \\
\text{Case } e \text{ of } \\
\quad x \quad = \quad \ldots \\
\quad n \quad = \quad \ldots \\
\quad \lambda x. e \quad = \quad \ldots \\
\quad (e_1 \ e_2) \quad = \quad \ldots \\
\ldots
\]

This is just \( \Gamma \) (it’s hard to write \( \Gamma \) in code).
Simple Inference Algorithm (cont-1)

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

Case \( e \) of

\[ \Gamma \vdash N : \tau \] = \( \text{int} = \tau \)
\( N = (\emptyset, \text{Typeof}(N)) \)

\[ \Gamma \vdash x : \tau \] = \( \Gamma(x) = \tau \)
\( x = \text{if } (x \not\in \text{Dom}(TE)) \text{ then Fail } \)
\( \text{else let } \tau = TE(x); \)
\( \text{in } (\emptyset, \tau) \)

\[ \Gamma \vdash \lambda x. e : \tau \] = \( \exists a_1a_2. (\ [\Gamma; x : a_1 \vdash e : a_2] \land \tau = a_1 \rightarrow a_2 ) \)
\( \lambda x. e = \text{let } (S_1, \tau_2) = W(TE + \{ x : u \}, e) \)
\( \text{in } (\ , \ ) \)

\[ \Gamma \vdash e_1e_2 : \tau \] = \( \exists a. (\ [\Gamma \vdash e_1 : a \rightarrow \tau] \land [\Gamma \vdash e_2 : a] ) \)
\( (e_1 e_2) = \text{let } (S_1, \tau_1) = W(TE, e_1); \)
\( (S_2, \tau_2) = W(S_1(TE), e_2); \)
\( S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \rightarrow u); \)
\( \text{in } (S_3 S_2 S_1, S_3(u)) \)
Simple Inference Algorithm (cont-1)

\[
\text{Def } W(\text{TE}, e) = \\
\text{Case } e \text{ of} \\
\quad c = (\emptyset, \text{Typeof}(c)) \\
\quad x = \begin{cases} \\
\quad \text{if } (x \not\in \text{Dom}(\text{TE})) \text{ then Fail} \\
\quad \text{else let } \tau = \text{TE}(x); \quad \\
\quad \text{in } (\emptyset, \tau) \\
\quad \lambda x.e = \begin{cases} \\
\quad \text{let } (S_1, \tau_1) = W(\text{TE} + \{ x : u \}, e) \\
\quad \text{in } (S_1, S_1(u) \rightarrow \tau_1) \\
\quad (e_1 \ e_2) = \begin{cases} \\
\quad \text{let } (S_1, \tau_1) = W(\text{TE}, e_1); \\
\quad (S_2, \tau_2) = W(S_1(\text{TE}), e_2); \\
\quad S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \rightarrow u); \\
\quad \text{in } (S_3 \ S_2 \ S_1, S_3(u)) \\
\quad \text{let } x = e_1 \text{ in } e_2 \\
\quad = \begin{cases} \\
\quad \text{let } (S_1, \tau_1) = W(\text{TE} + \{ x : u \}, e_1); \\
\quad S_2 = \text{Unify}(S_1(u), \tau_1); \\
\quad (S_3, \tau_2) = W(S_2 \ S_1(\text{TE}) + \{ x : \tau_1 \}, e_2); \\
\quad \text{in } (S_3 \ S_2 \ S_1, \tau_2)
\end{cases}
\end{cases}
\end{aligned}
\]
Example

\[ W(\emptyset, \lambda f. f \, 5) = (\emptyset, \text{Int}) \]

\[ W(\{f: u_0\}, f) = (\emptyset, u_0) \]

\[ W(\{f: u_0\}, f \, 5) = (\emptyset, \text{Int}) \]

\[ W(\emptyset, (\lambda f. f \, 5)) = (\emptyset, \text{Int}) \]

\[ W(\emptyset, (\lambda f. f \, 5)(\lambda x. x)) = (\emptyset, \text{Int}) \]
Example

\[ W(\{f: u_0\}, f) = (\emptyset, u_0) \]
\[ W(\{f: u_0\}, 5) = (\emptyset, \text{Int}) \]

\[ \text{Unify}(u_0, \text{Int} \rightarrow u_1) = [(\text{Int} \rightarrow u_1)/u_0] \]

\[ W(\{f: u_0\}, f \ 5) = \]

\[ W(\emptyset, (\lambda f. \ f \ 5)) = \]

\[ W(\emptyset, (\lambda f. \ f \ 5)(\lambda x. x)) \]
Example

\[ \text{Def } W(\text{TE}, e) = \]

\[ \text{Case } e \text{ of} \]

\[ \ldots \]

\[ \lambda x. e = \text{let } (S_1, \tau_1) = W(\text{TE} + \{ x : u \}, e) \]
\[ \text{in } (S_1, S_1(u) \to \tau_1) \]

\[ (e_1 \; e_2) = \text{let } (S_1, \tau_1) = W(\text{TE}, e_1); \]
\[ (S_2, \tau_2) = W(S_1(\text{TE}), e_2); \]
\[ S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \to u); \]
\[ \text{in } (S_3 \; S_2 \; S_1, S_3(u)) \]

\[ W(\{f:u_0\}, f) = (\emptyset, u_0) \quad W(\{f:u_0\}, 5) = (\emptyset, \text{Int}) \]

\[ \text{Unify}(u_0, \text{Int} \to u_1) = [(\text{Int} \to u_1)/u_0] \]

\[ W(\{f:u_0\}, f \; 5) = ([\text{Int} \to u_1/u_0], u_1) \]

\[ W(\emptyset, (\lambda f. f \; 5)) = [(\text{Int} \to u_1)/u_0], (\text{Int} \to u_1) \to u_1) \]

\[ W(\emptyset, (\lambda f. f \; 5)(\lambda x. x)) \]
Example

\[ W(\emptyset, (\lambda f. f \, 5)) = \left(\left(\text{Int} \rightarrow u_1\right)/u_0, (\text{Int} \rightarrow u_1) \rightarrow u_1\right) \]

\[ W(\emptyset, (\lambda x. x)) = (\emptyset, u_3 \rightarrow u_3) \]

\[ \text{Unify}((\text{Int} \rightarrow u_1) \rightarrow u_1, (u_3 \rightarrow u_3) \rightarrow u_4) = \]

\[ W(\emptyset, (\lambda f. f \, 5)(\lambda x. x)) \]
Example

\[
de\text{f Unify}(\tau_1, \tau_2) = \begin{cases} 
(\tau_1, t_2) & = [\tau_1 / t_2] \text{ provided } t_2 \not\in \text{FV}(\tau_1) \\
(t_1, \tau_2) & = [\tau_2 / t_1] \text{ provided } t_1 \not\in \text{FV}(\tau_2) \\
(\iota_1, \iota_2) & = \text{if (eq? } \iota_1 \iota_2) \text{ then [ ] else fail} \\
(\tau_{11} \rightarrow \tau_{12}, \tau_{21} \rightarrow \tau_{22}) & = \text{let } S_1 = \text{Unify}(\tau_{11}, \tau_{21}) \\
& \text{S}_2 = \text{Unify}(S_1(\tau_{12}), S_1(\tau_{22})) \\
in & S_2 S_1
\end{cases}
\]

\[
\text{Unify}((\text{Int} \rightarrow u_1), (u_3 \rightarrow u_3)) = [\text{Int} / u_3 ; \text{Int} / u_1]
\]

\[
\text{Unify}((\text{Int} \rightarrow u_1) \rightarrow u_1, (u_3 \rightarrow u_3) \rightarrow u_4) = [\text{Int} / u_3 ; \text{Int} / u_1 ; \text{Int} / u_4]
\]

\[
W(\emptyset, (\lambda f. f 5)(\lambda x. x))
\]
Example

\[ W(\emptyset, (\lambda f. f \, 5)) = ([\text{Int} \to _u u_1]/u_0, \text{Int} \to _u u_1) \to _u u_1 \]
\[ W(\emptyset, (\lambda x. x)) = (\emptyset, u_3 \to _u u_3) \]
\[ \text{Unify}((\text{Int} \to _u u_1) \to _u u_1, (u_3 \to _u u_3) \to _u u_4) = [\text{Int}/u_3; \text{Int}/u_1; \text{Int}/u_4] \]
\[ W(\emptyset, (\lambda f. f \, 5)(\lambda x. x)) = ([\text{Int} \to _u u_1]/u_0; \text{Int}/u_3; \text{Int}/u_1; \text{Int}/u_4], \text{Int}) \]
What about Let?

- \textit{let} \( x = e_1 \text{ in } e_2 \)

- **Typing rule**

\[
\frac{\Gamma; x : \tau' \vdash e_1 : \tau' \quad \Gamma; x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}
\]

- **Constraints**

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau = \exists \tau', \ (\Gamma; x : \tau' \vdash e_1 : \tau') \land (\Gamma; x : \tau' \vdash e_2 : \tau)
\]

- **Algorithm**

\[
\text{Case Exp = let } \ x = e_1 \text{ in } e_2
\Rightarrow \text{let } (S_1, \tau_1) = W(TE + \{x : u\}, e_1);
\]
\[
S_2 = \text{Unify}(S_1(u), \tau_1);
\]
\[
(S_3, \tau_2) = W(S_2 S_1(TE) + \{x : \tau_1\}, e_2);
\]
\[
in \ (S_3 S_2 S_1, \tau_2)
\]

This is Hindley Milner without polymorphism.
Polymorphism
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

```
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*
Explicit polymorphism

- You’ve seen this before

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

- How do we formalize this?

\[
\begin{align*}
\Gamma \vdash e : \tau \\
\hline
\Gamma \vdash \Lambda t. e : \forall t. \tau \\
\Gamma \vdash e[\tau] : \tau'[\tau / t]
\end{align*}
\]

- Example

\[
id = \Lambda T. \lambda x : T. x \\
id[\text{int}] 5
\]
Different Styles of Polymorphism

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

• 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!!
  – \((\lambda s: \forall T. \tau \ldots s[int]x \ldots s[bool]x)(\Lambda T. \text{code for sort})\)
Let polymorphism

- Introduce `let x = e1 in e2`
  - Just like saying `(\x.e2)e1`
  - Except `x` can be polymorphic

- Good engineering compromise
  - Enhance expressiveness
  - Preserve decidability

- This is the Hindley Milner type system
Type inference with polymorphism
Polymorphic Types

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

Constraints:

id :: t₁ --> t₁
id :: Int --> t₂
id :: Bool --> t₃

Does not unify!!

Solution: Generalize the type variable

id :: \(\forall t₁. t₁ --> t₁\)

Different uses of a generalized type variable may be instantiated differently

id₂ : Bool --> Bool
id₁ : Int --> Int

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

There are no types in the syntax of the language!

The type of each subexpression is derived by the Hindley-Milner type inference algorithm.

**Expressions**

\[
E ::= c \quad \text{constant} \\
| \ x \quad \text{variable} \\
| \ \lambda x. \ E \quad \text{abstraction} \\
| \ (E_1 \ E_2) \quad \text{application} \\
| \ \text{let } x = E_1 \ in \ E_2 \quad \text{let-block}
\]
A Formal Type System

Types
\[ \tau ::= t \]
\[ | \cdot \]
\[ | \tau_1 \rightarrow \tau_2 \]

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

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

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 \textit{instantiated} into a type \( \tau' \) by substituting types for \textit{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 \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*

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

### (App)

\[
\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau'}{\Gamma \vdash (e_1 e_2) : \tau'} \quad \text{Remember, } \tau \text{ stands for a monotype, } \sigma \text{ for a polymorphic type}
\]

### (Abs)

\[
\frac{\Gamma \vdash \{x:\tau\} e : \tau'}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau'}
\]

\(x\) can be considered of type \(\tau\) as long as its type as specified in the environment can be specialized to \(\tau\) (i.e. \(\tau\) is an instance of \(\sigma\)).

### (Var)

\[
\frac{(x:\sigma) \in \Gamma \quad \sigma \geq \tau}{\Gamma \vdash x : \tau}
\]

Note: \(x\) has a different type in \(e_1\) than in \(e_2\). In \(e_1\), \(x\) is not a polymorphic type, but in \(e_2\) it gets generalized into one.

### (Const)

\[
\frac{\text{typeof}(c) \geq \tau}{\Gamma \vdash c : \tau}
\]

### (Let)

\[
\frac{\Gamma \vdash \{x:\tau\} e_1 : \tau \quad \Gamma \vdash \{x:Gen(\Gamma,\tau)\} e_2 : \tau'}{\Gamma \vdash \text{(let } x=e_1 \text{ in } e_2) : \tau'}
\]
HM Inference Algorithm

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

c \quad = (\{\}, \text{Typeof}(c))

x \quad = \text{if} \ (x \notin \text{Dom}(TE)) \text{ then } \text{Fail}

\text{else let } \forall t_1...t_n. \tau = TE(x); \ \\ \quad \quad \quad \in \ (\{\}, [u_i/ t_i] \tau)

\lambda x.e \quad = \text{let } (S_1, \tau_1) = W(TE + \{x : u\}, e); \ \\ \quad \quad \quad \in (S_1, S_1(u) -> \tau_1)

(e_1 e_2) \quad = \text{let } (S_1, \tau_1) = W(TE, e_1); \ \\ \quad \quad \quad (S_2, \tau_2) = W(S_1(TE), e_2); \ \\ \quad \quad \quad S_3 = \text{Unify}(S_2(\tau_1), \tau_2 -> u); \ \\ \quad \quad \quad \in (S_3 S_2 S_1, S_3(u))

let \ x = e_1 \text{ in } e_2 \ 
\quad = \text{let } (S_1, \tau_1) = W(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(TE), S_2(\tau_1)); \ 
\quad \quad \quad (S_3, \tau_2) = W(S_2 S_1(TE) + \{x : \sigma\}, e_2); \ 
\quad \quad \quad \in (S_3 S_2 S_1, \tau_2)

u’s represent new type variables
Hindley-Milner: Example

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

\[ W(\emptyset, A) = ( \{\} , u_1 \rightarrow (u_1, u_1) ) \]
\[ W(\{x : u_1\}, B) = ( \{\} , (u_1, u_1) ) \]
\[ W(\{x : u_1, f : u_2\}, \lambda y. x) = ( \{\} , u_3 \rightarrow u_1 ) \]
\[ W(\{x : u_1, f : u_2, y : u_3\}, x) = ( \{\} , u_1 ) \]

Unify(u_2, u_3 \rightarrow u_1) = [ (u_3 \rightarrow u_1) / u_2 ]

Gen(\{x : u_1\}, u_3 \rightarrow u_1) = \forall u_3. u_3 \rightarrow u_1

TE = \{x : u_1, f : \forall u_3. u_3 \rightarrow u_1\}

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

W(TE, f) = ( \{\} , u_4 \rightarrow u_1 )

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

Unify(u_4 \rightarrow u_1, \text{Int} \rightarrow 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 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.

\[
\text{let}
\begin{align*}
twice\ f\ x &= f\ (f\ x) \\
\text{in}
\end{align*}
\begin{align*}
twice\ twice\ succ\ 4
\end{align*}
\]

versus

\[
\text{let}
\begin{align*}
twice\ f\ x &= f\ (f\ x) \\
\text{foo}\ g &= (g\ g\ succ)\ 4 \\
\text{in}
\end{align*}
\begin{align*}
\text{foo}\ twice
\end{align*}
\]

foo is not type correct!

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

(Var)  \[(x : \tau) \in TE\]  \[\infer[TE] {x : \tau} \]

(Spec)  \[TE \vdash e : \forall t.\tau\]  \[\infer[TE] {e : \tau [u/t]} \]

(Gen)  \[TE \vdash e : \tau \quad t \notin FV(TE)\]  \[\infer[TE] {e : \forall t.\tau} \]

(Let)  \[TE+\{x:\tau\} \vdash e_1 : \tau \quad \infer[TE+\{x:\tau\}] {e_2 : \tau'} \]  \[\infer[TE] {\text{(let } x = e_1 \text{ in } e_2 : \tau')} \]

(App) and (Abs) rules remain unchanged.

Sound but no inference algorithm!
6.820 Fundamentals of Program Analysis
Fall 2015

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.