Types and Simple Type Inference

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

September 21, 2006

Outline

- General issues
- Type instances
- Type Unification
- Type Inference rules for a simple non-polymorphic type system
  - Type Inference rules for a polymorphic type system
  - Overloading next time ...
What are Types?

- A method of classifying objects (values) in a language

\[ x :: \tau \]

says object \( x \) has type \( \tau \) or object \( x \) belongs to a type \( \tau \)

- \( \tau \) denotes a set of values.

This notion of types is different from types in languages like C, where a type is a storage class specifier.

Type Correctness

- If \( x :: \tau \) then only those operations that are appropriate to set \( \tau \) may be performed on \( x \).

- A program is type correct if it never performs a wrong operation on an object.

  - Add an \textit{Int} and a \textit{Bool}
  - Head of an \textit{Int}
  - Square root of a \textit{list}
Type Safety

- A language is type safe if only type correct programs can be written in that language.

- Most languages are not type safe, i.e., have “holes” in their type systems.
  
  *Fortran:* Equivalence, Parameter passing
  *Pascal:* Variant records, files
  *C, C++:* Pointers, type casting

  However, Java, CLU, Ada, ML, Id, Haskell, pH etc. are type safe.

Type Declaration vs Reconstruction

- Languages where the user must declare the types
  - CLU, Pascal, Ada, C, C++, Fortran, Java

- Languages where type declarations are not needed and the types are reconstructed at run time
  - Scheme, Lisp

- Languages where type declarations are generally not needed but allowed, and types are reconstructed at compile time
  - ML, Id, Haskell, pH

A language is said to be statically typed if type-checking is done at compile time.
Polymorphism

- In a monomorphic language like Pascal, one defines a different length function for each type of list.

- In a polymorphic language like ML, one defines a polymorphic type (list t), where t is a type variable, and a single function for computing the length.

- pH and most modern functional languages have polymorphic objects and follow the Hindley-Milner type system.

Type Instances

The type of a variable can be instantiated differently within its lexical scope.

```ml
let id = \x.x
in ((id1 5), (id2 True))

id1 :: Int --> Int
id2 :: Bool --> Bool

Both id1 and id2 can be regarded as instances of type

t --> t
```
Type Instances: another example

```
let
  twice :: (t -> t) -> t -> t
  twice f x = f (f x)
in
  twice1 twice2 (plus 3) 4
```

\[ \text{twice}_1 :: (\text{I} \rightarrow \text{I}) \rightarrow \text{I} \rightarrow \text{I} \rightarrow (\text{I} \rightarrow \text{I}) \rightarrow (\text{I} \rightarrow \text{I}) \]

\[ \text{twice}_2 :: (\text{I} \rightarrow \text{I}) \rightarrow \text{I} \rightarrow \text{I} \]

Type Instantiation:
\(\lambda\)-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
```

vs.

```
let
  twice f x = f (f x)
in
  foo twice
```

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

Generic vs. Non-generic type variables
A mini Language (\(\lambda\)-calculus + let) 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.

*but first a Simple Type System* ...

A Simple Type System

**Expressions**

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

**Types**

\[
\tau ::= \iota \quad \text{base types} \\
| t \quad \text{type variables} \\
| \tau_1 \rightarrow \tau_2 \quad \text{Function types}
\]

**Type Environments**

\[
TE ::= \text{Identifiers} \rightarrow \text{Types}
\]
Type Inference Issues

- 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 \( \tau_a = \tau_b \)?
    
    This is true if \( \tau_1 = \tau_3 \) 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**

**Types**

\[
\tau ::= \iota \quad \text{base types (Int, Bool ..)} \\
| t \quad \text{type variables} \\
| \tau_1 \rightarrow \tau_2 \quad \text{Function types}
\]

A substitution is a map

\[ S : \text{Type Variables} \rightarrow \text{Types} \]

\[ S = [\tau_1 / t_1, ..., \tau_n / t_n] \]

\( S' = S \upharpoonright \tau \)

\( S' \) is a **Substitution Instance of** \( \tau \)

**Example:**

\[ S = [(t \rightarrow \text{Bool}) / t_1] \]

\[ S(t_1 \rightarrow t_1) = (t \rightarrow \text{Bool}) \rightarrow (t \rightarrow \text{Bool}) \]

Substitutions can be **composed**, i.e., \( S_2 S_1 \)

**Example:**

\[ S_1 = [(t \rightarrow \text{Bool}) / t_1] ; S_2 = [\text{Int} / t] \]

\[ S_2 S_1 (t_1 \rightarrow t) = (\text{Int} \rightarrow \text{Bool}) \rightarrow (\text{Int} \rightarrow \text{Bool}) \]
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

\[
def \text{Unify}(\tau_1, \tau_2) = \begin{cases} 
(\tau_1, \tau_2) = [\tau_1 / \tau_2] & \text{provided } \tau_2 \not\in \text{FV}(\tau_1) \\
(\tau_1, \tau_2) = [\tau_2 / \tau_1] & \text{provided } \tau_1 \not\in \text{FV}(\tau_2) \\
(\iota_1, \iota_2) = \text{if (eq? } \iota_1 \iota_2) \text{ then [ ]} & \\
\text{else fail} \\
(\tau_{1\rightarrow}\tau_{12}, \tau_{21} \rightarrow \tau_{22}) \\
= \text{let } S_1=\text{Unify}(\tau_{11}, \tau_{21}) \\
S_2=\text{Unify}(S_1(\tau_{12}), S_1(\tau_{22})) \\
\text{in } S_2 S_1 \\
\text{otherwise } = \text{fail}
\end{cases}
\]

Does the order matter?

Type Inference Rules

Typing: \(\text{TE } \vdash e : \tau\)

Suppose we want to assert (prove) that given some type environment \(\text{TE}\), the expression \((e_1 e_2)\) has the type \(\tau'\).

Then it must be the case that the same \(\text{TE}\) implies that \(e_1\) has type \(\tau'\) and \(e_2\) has the type \(\tau\).

Such an inference rule can be written as:

\[
\frac{\text{TE } \vdash e_1 : \tau \rightarrow \tau' \quad \text{TE } \vdash e_2 : \tau}{\text{TE } \vdash (e_1 e_2) : \tau'}
\]
Simple Type Inference Rules

Typing: \( \frac{\text{Typing: } \quad \text{TE} \vdash e : \tau}{\text{(App)}} \)

\[ \frac{\text{TE} \vdash e_1 : \tau \rightarrow \tau'} {\text{TE} \vdash (e_1 \; e_2) : \tau'} \]

\[ \frac{\text{TE} \vdash \lambda x. e : \tau \rightarrow \tau'} {\text{(Abs)}} \]

\[ \frac{\text{TE} \vdash x : \tau}{\text{(Var)}} \]

\[ \frac{\text{typeof}(c) = \tau}{\text{(Const)}} \]

\[ \frac{\text{TE} \vdash c : \tau}{\text{(Let)}} \]

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

Inference Algorithm

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

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

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

\[ \begin{align*}
  x &= \ldots \\
  \lambda x. e &= \ldots \\
  (e_1 \; e_2) &= \ldots \\
  \text{let } x = e_1 \; \text{in } e_2 &= \ldots
\end{align*} \]
Inference Algorithm (cont.)

Definition \( W(\text{TE}, e) \):

\[
\begin{align*}
\text{Def} \ W(\text{TE}, e) &= \text{Case } e \text{ of} \\
\text{x} &= \\
\text{if } (x \notin \text{Dom(TE)}) \text{ then Fail} \\
\text{else let } \tau = \text{TE}(x); \\
\text{in } (\{\}, \tau) \\
\lambda x.e &= \\
\text{let } (S_1, \tau_1) = W(\text{TE} + \{x : u\}, e); \\
\text{in } (S_1, S_1(u) \rightarrow \tau_1) \\
(e_1 \ e_2) &= \text{let } (S_1, \tau_1) = W(\text{TE}, e_1) \\
&S_2 = W(S_1(\text{TE}), e_2); \\
&S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \rightarrow u); \\
\text{in } (S_3 \ S_2 \ S_1, S_3(u)) \\
\text{let } x = e_1 \ \text{in } e_2 &= \\
\text{let } (S_1, \tau_1) = W(\text{TE} + \{x : u\}, e_1); \tau_2) = W(S_2(\text{TE}) + \{x : \tau_1\}, e_2); \\
\text{in } (S_3 \ S_2 \ S_1, \tau_2)
\end{align*}
\]

Type Inference

Let \( \text{fact} = \lambda n. \text{if } (n == 0) \text{ then } 1 \text{ else } n \ * \ \text{fact } (n-1) \)

In fact
Inferring Polymorphic Types

\[\text{let } \ \text{id} = \lambda x. x \ \text{in} \ \ldots \text{id True} \ldots \text{id 1} \ldots\]