Simple Types and Type Inference

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.
September 22, 2010

Outline

• General issues
• Type instances
• Type Unification
• Type Inference rules for a simple non-polymorphic type system
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 \( Int \) and a \( Bool \)
  - Head of an \( Int \)
  - Square root of a \( list \)
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, Ada, CLU, ML, Id, Haskell, Bluespec, 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, Bluespec

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.

- Haskell and most modern functional languages have polymorphic types and follow the Hindley-Milner type system.

Simple types = Non polymorphic types

more on polymorphic types – next time ...

---

\[ \lambda \text{-calculus with Constants & Letrec} \]

\[ E ::= x \mid \lambda x. E \mid E E \]
\[ \mid \text{Cond} (E, E, E) \]
\[ \mid \text{PF}_k(E_1, \ldots, E_k) \]
\[ \mid \text{CN}_0 \]
\[ \mid \text{CN}_k(E_1, \ldots, E_k) \mid \text{CN}_k(SE_1, \ldots, SE_k) \]
\[ \mid \text{let } S \text{ in } E \]

\[ \text{PF}_1 ::= \text{negate} \mid \text{not} \mid \ldots \mid \text{Prj}_1 \mid \text{Prj}_2 \mid \ldots \]
\[ \text{PF}_2 ::= + \mid \ldots \]
\[ \text{CN}_0 ::= \text{Number} \mid \text{Boolean} \]
\[ \text{CN}_2 ::= \text{cons} \mid \ldots \]

Statements

\[ S ::= \varepsilon \mid x = E \mid S; S \]

There are no types in the syntax of the language!
A Simple Type System

Types
\[ \tau ::= \iota \quad | \quad \tau \quad | \quad \tau_1 \rightarrow \tau_2 \]

base types

<table>
<thead>
<tr>
<th>type variables</th>
</tr>
</thead>
</table>

Function types

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

int, bool, ...

Type Inference Preliminaries

- 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 \( \tau_a = \tau_b \)?

- 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?
    Suppose \( \tau_a = t_1 \rightarrow \text{Bool} \)
    \( \tau_b = \text{Int} \rightarrow \text{Int} \)
    Are \( \tau_a \) and \( \tau_b \) unifiable?
Simple Type Substitutions
needed to define type unification

Types

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

A substitution is a map
\[ S : \text{Type Variables} \rightarrow \text{Types} \]
\[ S = [\tau_1 / t_1, \ldots, \tau_n / t_n] \]
\[ \tau' = S \tau \quad \text{\(\tau'\) is a Substitution Instance of \(\tau\)} \]

Example:
\[ S = [(t \to \text{Bool}) / t_1] \]
\[ S (t_1 \to t_1) = ? \]

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

Example:
\[ S_1 = [(t \to \text{Bool}) / t_1] ; S_2 = [\text{Int} / t] \]
\[ S_2 S_1 (t_1 \to t_1) = ? \]

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

\[
\text{def Unify}(\tau_1, \tau_2) = \\
\text{case } (\tau_1, \tau_2) \text{ of} \\
(\tau_1, \tau_2) = [\tau_1 / \tau_2] \text{ provided } t_2 \notin \text{FV}(\tau_1) \\
(\tau_1, \tau_2) = [\tau_2 / \tau_1] \text{ provided } t_1 \notin \text{FV}(\tau_2) \\
(\iota_1, \iota_2) = \text{if } (\text{eq? } \iota_1 \iota_2) \text{ then } [ ] \text{ else fail } \\
(\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22})
\]
Type Inference

• Type inference is typically presented in two different forms:
  - Type inference rules: Rules define a way to deduce the type of each expression in terms of its environment and the types of its subexpressions
    • Clean and concise; needed to study the semantic properties, i.e., soundness, of the type system
  - Type inference algorithm: Needed by the compiler writer to deduce the type of each subexpression or to deduce that the expression is ill typed.

• Sometimes it is difficult to derive an inference algorithm for a given set of type inference rules.

Type Inference Rules

Typing: \[ \text{TE} \vdash e : \tau \]

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

Then it must be the case that the same TE implies that there exists some a type \(\tau\), s.t. \(e_1\) has type \(\tau \rightarrow \tau'\) and \(e_2\) has the type \(\tau\).

Such an inference rule can be written as:

\[
\text{(App) } \quad \text{TE} \vdash e_1 : \tau_1 \quad \text{TE} \vdash e_2 : \tau_2 \quad \text{TE} \vdash (e_1 \ e_2) : \tau'
\]
Simple Type Inference Rules

Typing: \[ \text{TE} \vdash e : \tau \]

(App) \[ \begin{array}{c} \text{TE} \vdash e_1 : \tau \to \tau' \\ \text{TE} \vdash e_2 : \tau \\ \text{TE} \vdash (e_1 \ e_2) : \tau' \end{array} \]

(Abs) \[ \begin{array}{c} \text{TE} \vdash \lambda x. e : \tau \to \tau' \end{array} \]

(Var) \[ \begin{array}{c} \text{TE} \vdash x : \tau \end{array} \]

(Const) \[ \begin{array}{c} \text{TE} \vdash c : \tau \end{array} \]

(Let) \[ \begin{array}{c} \text{TE} \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau' \end{array} \]

Simple Type Inference Rules \textit{cont}

(Cond) \[ \begin{array}{c} \text{TE} \vdash \text{Cond}(e_B, e_1, e_2) : \tau \end{array} \]

(PF) \[ \begin{array}{c} \text{TE} \vdash (e_1 + e_2) : \text{int} \end{array} \]

(CN) \[ \begin{array}{c} \text{TE} \vdash \text{CN}(e_1, e_2) : \text{CN}(\tau_1, \tau_2) \end{array} \]

(Pro) \[ \begin{array}{c} \text{TE} \vdash \text{Prj}_1(e) : \tau_1 \end{array} \]
Simple Inference Algorithm

\[ W(\text{TE}, e) \] returns \((S, \tau)\) 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.

**Def**

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

\[
\text{Case } e \text{ of } \\
\quad x = \ldots \\
\quad \lambda x.e = \ldots \\
\quad (e_1 e_2) = \ldots \\
\quad \text{let } x = e_1 \text{ in } e_2 = \ldots \\
\quad \text{Plus}(e_1, e_2) = \ldots \\
\quad \ldots
\]

Simple Inference Algorithm *(cont-1)*

**Def**

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

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

September 22, 2010 http://stellar.mit.edu/S/course/6/fa10/6.820 L05-17
Simple Inference Algorithm (cont-1)

Cond($e_B, e_1, e_2$) = \( \text{let } (S_B, \tau_B) = W(TE, e_B); (S_1, \tau_1) = W(S_B(TE), e_1); (S_2, \tau_2) = W(S_1 S_B(TE), e_2); S_3 = \text{Unify}(\tau_B, \text{bool}); S_4 = \text{Unify}(\tau_2, \tau_2); \in (S_4 S_3 S_2 S_1 S_B, S_4 S_3 \tau_2) \)

Plus($e_1, e_2$) = \( \text{let } (S_1, \tau_1) = W(TE, e_1); (S_2, \tau_2) = W(S_1(TE), e_2); S_3 = \text{Unify}(\tau_1, \text{int}); S_4 = \text{Unify}(\tau_2, \text{int}); \in (S_4 S_3 S_2 S_1, \text{int} \rightarrow \text{int}) \)

CN($e_1, e_2$) = \( \text{let } (S_1, \tau_1) = W(TE, e_1); (S_2, \tau_2) = W(S_1(TE), e_2); \in (S_2 S_1, \text{CN}(\tau_1, \tau_2)) \)

Prj$_1$(e) = \( \text{let } (S, \tau) = W(TE, e); S_1 = \text{Unify}(\tau, \text{CN}(u_1, u_2))); \in (S S_1 S_1(u_1)) \)

Type Inference : Example

\text{let } f = \lambda n. \text{cond } ((\text{eq0 } n), 1, n \ast (f \ (\text{pred } n)) \) \text{ in } f \)

\begin{align*}
W(\emptyset, A) &= \\
W(\{f : u_1\}, \lambda n.B) &= \\
W(\{f : u_1, n : u_2\}, B) &= \\
W(\{f : u_1, n : u_2\}, (\text{eq0 } n)) &=
\end{align*}
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: $TE \vdash e : \tau$ and $(e \rightarrow e') \Rightarrow TE \vdash e' : \tau$

  Progress: $TE \vdash e : \tau \Rightarrow$
  
  Either $e$ is a value or $\exists e'$ s.t. $(e \rightarrow e')$

Proofs in the next lecture

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