The Hindley-Milner Type System

Arvind
Laboratory for Computer Science
M.I.T.

September 25, 2002

http://www.csg.lcs.mit.edu/6.827

Outline

• General issues
• Type instances
• Type Unification
• Type Generalization
• A formal 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 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.

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

id₁ :: ?
id₂ :: ?
```

Both id₁ and id₂ can be regarded as instances of type ?
Type Instances: another example

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

```
twice1 :: ?

  twice2 :: ?
```

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 (plus 3) 4
```

```
VS.

let
  twice f x = f (f x)
  foo g = (g g (plus 3)) 4
in
  foo twice
```

Generic vs. Non-generic type variables
A mini Language
to study Hindley-Milner Types

Expressions

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

- There are no types in the syntax of the language!
- The type of each subexpression is derived by the Hindley-Milner type inference algorithm.

Types

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

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 \( \tau_a = \tau_b \)?

- 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

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

A substitution is a map
S : Type Variables --> Types

\[ S = [\tau_1 \rightarrow t_1, \ldots, \tau_n \rightarrow t_n] \]

\[ \tau' = S \tau \]
\n\tau' is a Substitution Instance of \tau

Example:
\[ S = [(t \rightarrow \text{Bool}) / t_1] \]
\[ S( t_1 \rightarrow \text{t}_1) = ? \]

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

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

\[ S_2 \circ S_1 \ ( t_1 \rightarrow \text{t}_1) = ? \]

Unification

An essential subroutine for type inference

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

\[ \text{def Unify}(\tau_1, \tau_2) = \]
\[ \text{case } (\tau_1, \tau_2) \text{ of} \]
\[ (\tau_1, \tau_2) = [\tau_1 / \tau_2] \]
\[ (t_1, t_2) = \text{if } (\text{eq? } t_1, t_2) \text{ then } [ ] \text{ else fail} \]
\[ (\tau_1 \rightarrow \tau_2, \tau_2 \rightarrow \tau_2) = \text{let } S_1=\text{Unify}(\tau_{11}, \tau_{21}) \]
\[ S_2=\text{Unify}(S_1(\tau_{12}), S_1(\tau_{22})) \]
\[ \text{in } S_2 S_1 \]
\[ \text{otherwise } = \text{fail} \]

Does the order matter?
Inferring Polymorphic Types

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

Constraints:

\[
\begin{align*}
\text{id} & : t_1 \rightarrow t_1 \\
\text{id} & : \text{Int} \rightarrow t_2 \\
\text{id} & : \text{Bool} \rightarrow t_3
\end{align*}
\]

Solution: Generalize the type variable:

\[
\text{id} : \forall t_1. t_1 \rightarrow t_1
\]

Different uses of a generalized type variable may be instantiated differently

\[
\begin{align*}
\text{id}_2 & : \text{Bool} \rightarrow \text{Bool} \\
\text{id}_1 & : \text{Int} \rightarrow \text{Int}
\end{align*}
\]

Generalization is Restricted

\[
f = \lambda g. \ldots (g \text{ True}) \ldots (g \text{ 1}) \ldots
\]

Can we generalize the type of \( g \) to \( \forall t_1, t_2. t_1 \rightarrow t_2 \) ?

There will be restrictions on \( g \) from the environment, the place of use, which may make this deduction unsound (incorrect).

Only generalize “new” type variables, the variables on which all the restrictions are visible.
A Formal Type System

**Types**
\[ \tau \ ::= \ i \ | \ t \ | \ \tau \rightarrow \tau \]

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

**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 \( \sigma \).

A type \( \tau \) is said to be **polymorphic** if it contains a type variable \( t \).

Example TE
\[
\{ + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \\
f :: \forall t. t \rightarrow t \rightarrow \text{Bool} \}
\]

Free and Bound Variables

**σ** = \( \forall \ t_1..t_n. \tau \)

**BV(σ)** = \{ \( t_1, ..., t_n \) \}

**FV(σ)** = \{ type variables of \( \tau \) \} - \{ \( t_1, ..., t_n \) \}

The definitions extend to Type Environments in an obvious way.

Example:
\[
\sigma ? = \forall t. (t_1 \rightarrow t_2)
\]

**FV(σ)** =
**BV(σ)** =
Type Substitutions

A substitution is a map
S : Type Variables --> Types
S = [τ_i/ t_1, t_2, ..., t_n]

τ' = S τ

τ' is a Substitution Instance of τ

σ' = S σ

Applied only to FV(σ), with renaming of BV(σ) as necessary

similarly for Type Environments

Examples:
S = [(t_2 --> Bool) / t_1]
S( t_1 --> t_1) = ( t_2 --> Bool) --> ( t_2 --> Bool)
S( ∀t_1.t_1 --> t_2) = ?
S( ∀t_2.t_1 --> t_2) = ?

Substitutions can be composed, i.e., S_2 S_1

Instantiations

• Type scheme σ can be instantiated into a type τ' by substituting types for BV(σ), that is,
  τ = S τ

- τ' is said to be an instance of σ (σ > τ')
- τ' is said to be a generic instance of σ when S maps variables to new variables.

Example:
σ = ∀t_1. t_1 --> t_2

a generic instance of σ is τ'}
Generalization *aka* Closing

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

- *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 \( \tau \)

Type Inference

- Type inference is typically presented in two different forms:
  - *Type inference rules*: Rules define the type of each expression
    - Needed for showing that the type system is *sound*
  - *Type inference algorithm*: Needed by the compiler writer to deduce the type of each subexpression or to deduce that the expression is ill typed.
- Often it is nontrivial to derive an inference algorithm for a given set of rules. There can be many different algorithms for a set of typing rules.

*next lecture* ...