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 τ?or object x belongs *to* a type τ?

- τ 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 τ 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, 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.

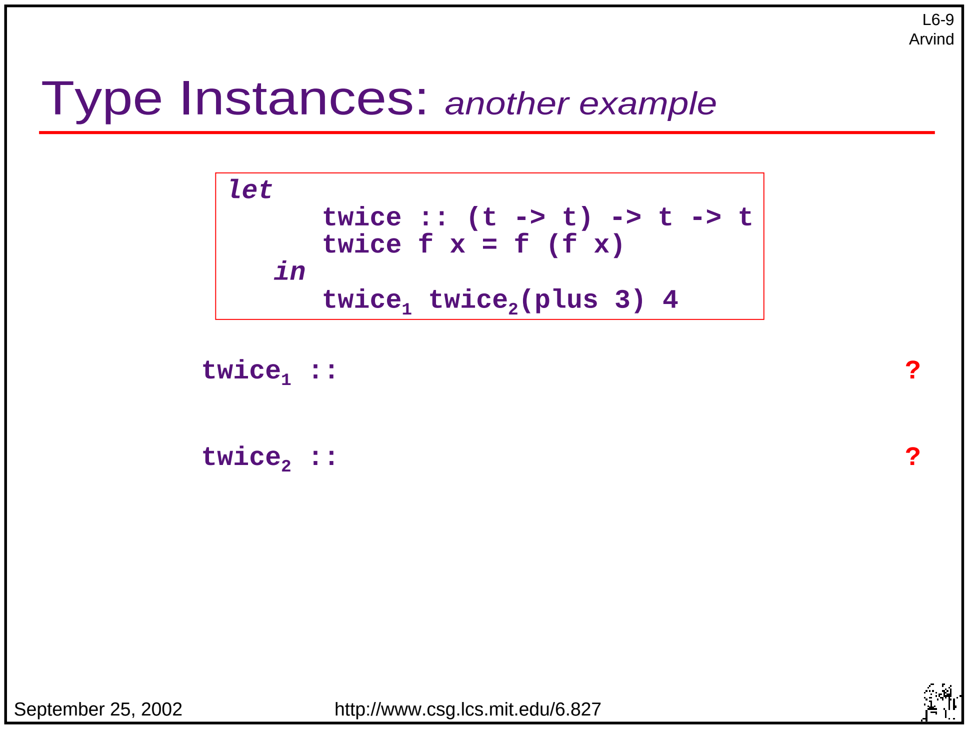
```
let
    id = \x.x
in
   ((id₁ 5), (id₂ True))
```

id_1 :: ?

id_2 :: ?

Both id_1 and id_2 can be regarded as instances of type

?



L6-9
Arvind

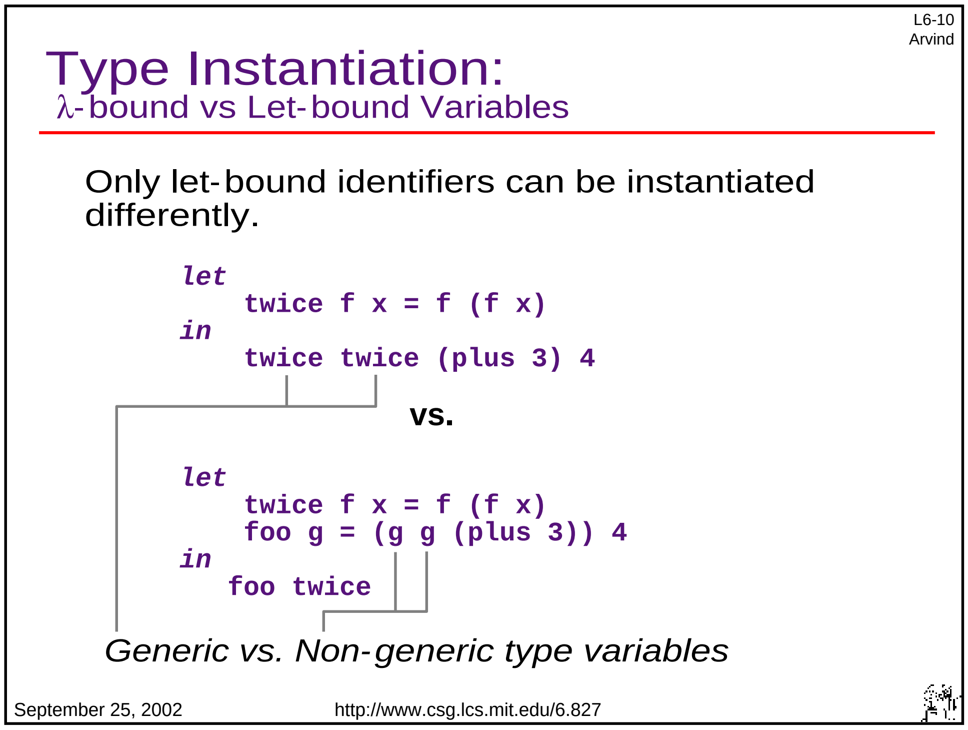
Type Instances: *another example*

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

$\texttt{twice}_1$ :: ?

$\texttt{twice}_2$ :: ?

September 25, 2002 http://www.csg.lcs.mit.edu/6.827



L6-10
Arvind

Type Instantiation:
λ-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

September 25, 2002 http://www.csg.lcs.mit.edu/6.827

A mini Language
to study Hindley-Milner Types

Expressions

E ::= c	constant
| x	variable
| ??. E	abstraction
| (E_1 E_2)	application
| *let* x = E_1 *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.*

Types

τ ::= ι	base types (Int, Bool ..)
| t	type variables
| τ_1 ?--> τ_2	Function types

Type Inference Issues

- What does it mean for two types τ_a ?and τ_b to be equal?
  - *Structural Equality*

  Suppose τ_a = τ_1 --> τ_2
  τ_b = τ_3 ?--> τ_4
  Is τ_a = τ_b ?

- Can two types be made equal by choosing appropriate substitutions for their type variables?
  - *Robinson's unification algorithm*

  Suppose τ_a = t_1 --> Bool
  τ_b = Int ?--> t_2
  Are τ_a and τ_b unifiable ?

  Suppose τ_a = t_1--> Bool
  τ_b = Int ?--> Int
  Are τ_a and τ_b unifiable ?

Simple Type Substitutions

Types

$\tau ::= \iota$	base types (Int, Bool ..)
$\mid\ t$	type variables
$\mid\ \tau_1 \text{ ?--> } \tau_2$	Function types

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

S = [$\tau_?$/ t_1,..., τ_n /? t_n]

τ' = S τ $\qquad$ τ' is a *Substitution Instance of* τ
Example:

S = [(t --> Bool) / t_1]
S(t_1 --> t_1) = $\qquad$?

Substitutions can be *composed,* i.e., S_2 S_1
Example:

S_1 = [(t --> Bool) / t_1] ; S_2 = [Int / t]

S_2 S_1 (t_1 --> t_1) = $\qquad$?

Unification

An essential subroutine for type inference

Unify(τ_1, τ_2) tries to unify τ_1 and τ_2 and returns a substitution if successful

```
def Unify(τ1, τ2) =
    case  (τ1, τ2) of
        (τ1, t2)  = [τ1 / t2]
        (t1, τ2)  = [τ2 / t1]
        (ι1, ι2)  = if (eq? ι1 ι2) then [ ]
                                   else  fail
(τ11--> τ12, τ21 --> τ22)
                  = let   S1=Unify(τ11, τ21)
                          S2=Unify(S1(τ12), S1(τ22))
                     in  S2 S1
    otherwise   = fail
```

Does the order matter?

Inferring Polymorphic Types

```
let
      id = λx. x
in
     ... (id True) ... (id 1) ...
```

Constraints:

id :: t_1 --> t_1
id :: Int --> t_2
id :: Bool --> t_3

Solution: Generalize the type variable:

id :: $\forall t_1$?? t_1 --> t_1

Different uses of a generalized type variable may be *instantiated* differently

id_2 : Bool --> Bool
id_1 : Int --> Int

Generalization is Restricted

f = λg. ...(g True) ... (g 1) ...

Can we generalize the type of g to ?

$\forall t_1\ t_2.\ t_1$ --> 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

$\tau? ::= \iota$	base types
$\mid$ t	type variables
$\mid \tau_1 \text{-->} \tau_2$	Function types

Type Schemes

$\sigma ::= \tau?$

$\mid \forall t.\ \sigma_?$

Type Environments

TE ::= Identifiers --> Type Schemes

Note, all the ∀'s occur in the beginning of a type scheme, i.e., a type τ cannot contain a type scheme σ

A type τ?is said to be *polymorphic* if it contains a type variable

Example TE { + :: Int --> Int --> Int,
f :: ∀ t. t --> t --> Bool }

Free and Bound Variables

$\sigma = ?\forall t_1..t_n.\ \tau$

$BV(\sigma) = \{ t_1,\ldots, t_n \}$

$FV(\sigma) = \{\text{type variables of } \tau\} - \{ t_1,\ldots, t_n \}$

The definitions extend to Type Environments in an obvious way

Example:

$\sigma ?= \forall t_1.\ (t_1 \text{-->} t_2)$

$FV(\sigma) =$

$BV(\sigma) =$

Type Substitutions

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

S = [$\tau_?$/ t_1????? τ_n?? t_n?]

$\tau' = S\ \tau$ τ' is a *Substitution Instance of* τ

$\sigma' = S\ \sigma$ 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($\forall t_1 . t_1$ --> t_2) = ?

S($\forall t_2 . t_1$ --> t_2) = ?

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

Instantiations

$$\boxed{\sigma = \forall\, t_1..t_n.\ \tau}$$

- Type scheme σ can be *instantiated* into a type τ' by substituting types for BV(σ), that is,
  $\tau' = S\ \tau$ for some S s.t. Dom(S) $\subseteq$ BV(σ)

  -? τ' is said to be an *instance of* σ ($\sigma > \tau'$)

  - τ' is said to be a *generic instance of* σ?when S maps variables to new variables.

Example:

$\sigma = \forall\, t_1 .\ t_1 \text{ --> } t_2$
a generic instance of σ?is ?

Generalization *aka Closing*

$$\boxed{\begin{array}{l}Gen(TE,\tau) = \forall t_1..t_n.\ \tau \\ \qquad\qquad\text{where } \{ t_1...t_n \} = FV(\tau) - FV(TE)\end{array}}$$

- *Generalization* introduces polymorphism
- Quantify type variables that are free in τ? but not *free* in the type environment (TE)
- Captures the notion of *new* type variables of τ

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 ...