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 τ or object x belongs *to* a type τ

- τ 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 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 :: $\boxed{\texttt{Int --> Int}}$?

id_2 :: $\boxed{\texttt{Bool --> Bool}}$?

Both id_1 and id_2 can be regarded as instances of type

$\boxed{\texttt{t --> t}}$?

Type Instances: *another example*

```
let
      twice :: (t -> t) -> t -> t
      twice f x = f (f x)
   in
```

      $\texttt{twice}_1$ $\texttt{twice}_2$ `(plus 3) 4`

$\texttt{twice}_1$ `::` `((I -> I) -> I -> I) -> (I -> I) -> (I -> I)` **?**

$\texttt{twice}_2$ `::` `(I -> I) -> I -> I` **?**

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 succ 4
```

vs.

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

foo is not type correct !

Generic vs. Non-generic type variables

A mini Language (λ-calculus + let)

to study Hindley-Milner Types

Expressions

E ::= c	constant
| x	variable
| λx. 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.*

but first a Simple Type System ...

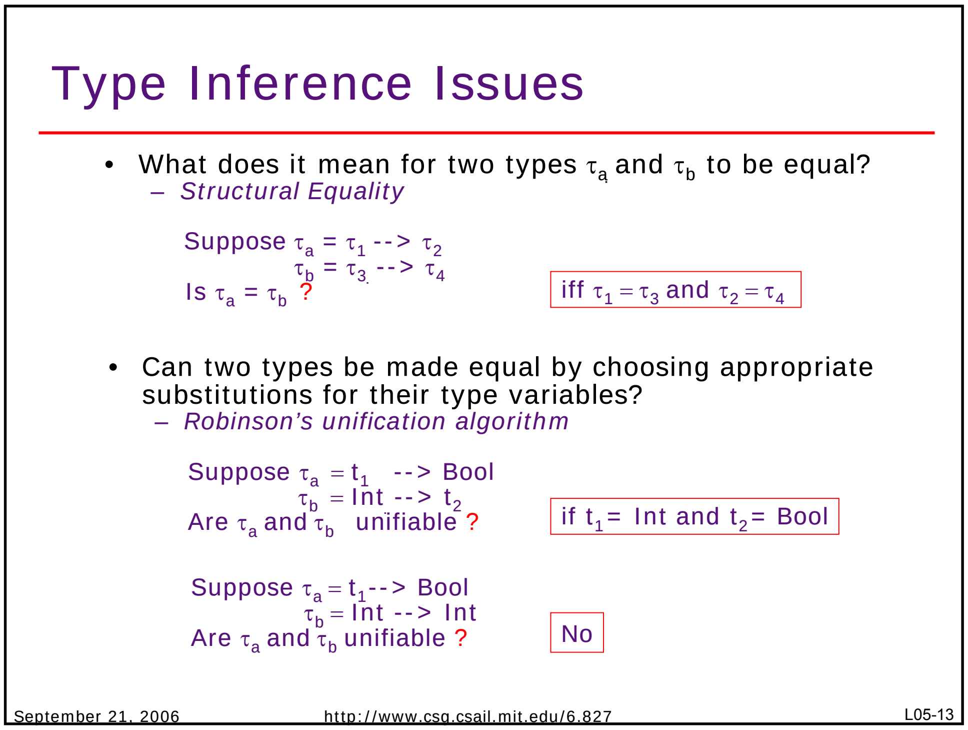
A Simple Type System

Types

τ ::= ι	base types
| t	type variables
| τ_1--> τ_2	Function types

Type Environments

TE ::= Identifiers --> Types



Type Inference Issues

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

Suppose $\tau_a = \tau_1 \dashrightarrow \tau_2$
$\tau_b = \tau_3 \dashrightarrow \tau_4$
Is $\tau_a = \tau_b$?

$\boxed{\text{iff } \tau_1 = \tau_3 \text{ and } \tau_2 = \tau_4}$

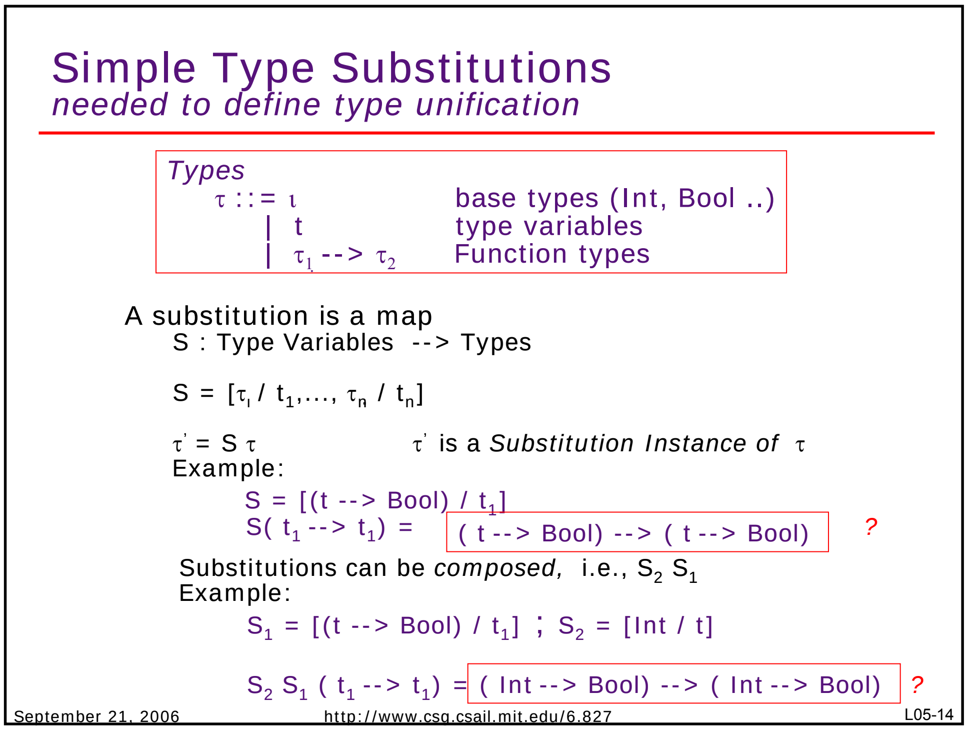
- Can two types be made equal by choosing appropriate substitutions for their type variables?
  - *Robinson's unification algorithm*

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

$\boxed{\text{if } t_1 = \text{Int and } t_2 = \text{Bool}}$

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

$\boxed{\text{No}}$



Simple Type Substitutions
needed to define type unification

Types

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

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

S = $[\tau_1 / t_1, \ldots, \tau_n / t_n]$

τ' = S τ τ' is a *Substitution Instance of* τ
Example:
S = [(t --> Bool) / t_1]
S(t_1 --> t_1) = $\boxed{\text{(t --> Bool) --> (t --> Bool)}}$?

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) = $\boxed{\text{(Int --> Bool) --> (Int --> Bool)}}$?

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($\tau_1$, $\tau_2$) =
    case ($\tau_1$, $\tau_2$) of
        ($\tau_1$, $t_2$) = [$\tau_1$ / $t_2$]  provided $t_2 \notin FV(\tau_1)$
        ($t_1$, $\tau_2$) = [$\tau_2$ / $t_1$]  provided $t_1 \notin FV(\tau_2)$
        ($\iota_1$, $\iota_2$) = if (eq? $\iota_1$ $\iota_2$) then [ ]
                                  else fail
($\tau_{11}$-->$\tau_{12}$, $\tau_{21}$-->$\tau_{22}$)
                    = let  $S_1$=Unify($\tau_{11}$, $\tau_{21}$)
                           $S_2$=Unify($S_1(\tau_{12})$, $S_1(\tau_{22})$)
                      in $S_2 S_1$
    otherwise    = fail
```

Does the order matter?

Type Inference Rules

Typing: TE |-- e : τ

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

Then it must be the case that the same TE implies that e_1 has type $\tau \dashrightarrow \tau'$ and e_2 has the type τ .

Such an inference rule can be written as:

$$\text{(App)}\quad \frac{TE \vdash e_1 : \tau \dashrightarrow \tau' \qquad TE \vdash e_2 : \tau}{TE \vdash (e_1\ e_2) : \tau'}$$

Simple Type Inference Rules

Typing: $TE \vdash e : \tau$

(App) $$\frac{TE \vdash e_1 : \tau \dashrightarrow \tau' \qquad TE \vdash e_2 : \tau}{TE \vdash (e_1\ e_2) : \tau'}$$

(Abs) $$\frac{\boxed{TE + \{x : \tau\} \vdash e : \tau'}}{TE \vdash \lambda x.e : \tau \dashrightarrow \tau'}$$

(Var) $$\frac{\boxed{(x : \tau)\ \varepsilon\ TE}}{TE \vdash x : \tau}$$

(Const) $$\frac{\boxed{\text{typeof}(c) = \tau}}{TE \vdash c : \tau}$$

(Let) $$\frac{\boxed{TE+\{x:\tau\} \vdash e_1 : \tau \qquad TE+\{x:\tau\} \vdash e_2 : \tau'}}{TE \vdash (\textit{let}\ x = e_1\ \textit{in}\ e_2) : \tau'}$$

Inference Algorithm

W(TE, e) returns (S,τ) such that S (TE) |-- e : τ

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

```
Def W(TE, e) =
    Case e of
      x                    =     ...
      λx.e                 =     ...
      (e1 e2)              =     ...
      let  x = e1 in e2    =     ...
```

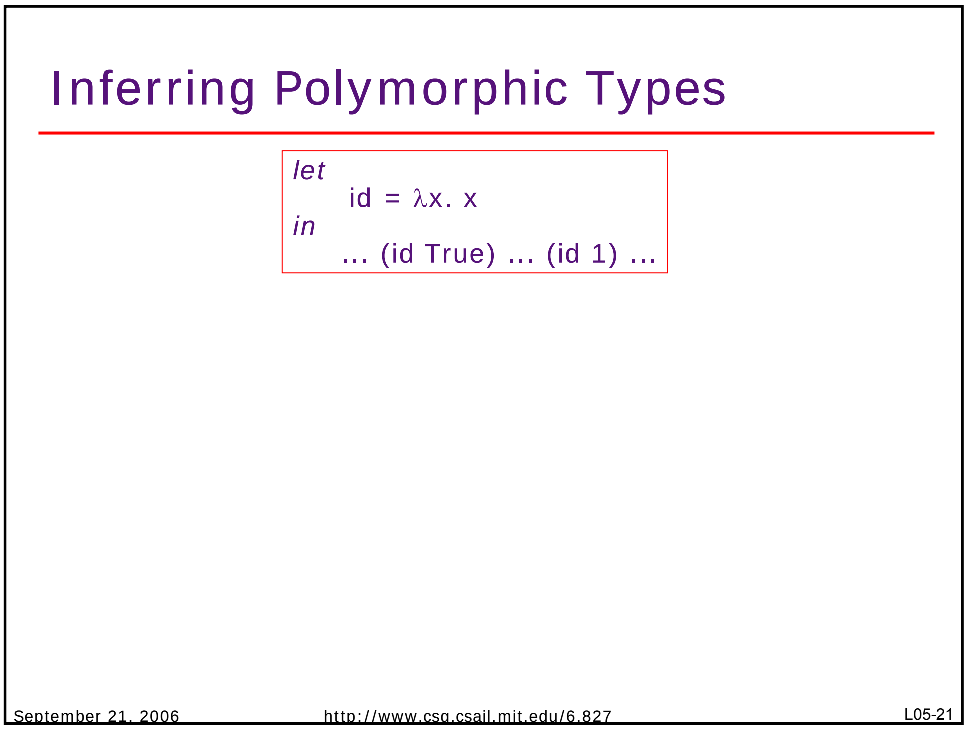
Inference Algorithm *(cont.)*

```
Def W(TE, e)  = Case e of
    x  =
        if  (x ∉ Dom(TE)) then Fail
                  else  let  τ  = TE(x);
                          in   ({}, τ )
    λx.e         =
        let  (S_1, τ_1) = W(TE + { x : u }, e);
        in   (S_1, S_1(u) --> τ_1)
    (e_1 e_2)      = let       (S_1, τ_1)  = W(TE, e_1)
                               (S_2, τ_2)  = W(S_1(TE), e_2)
                               S_3  = Unify(S_2(τ_1), τ_2 --> u);
                   in   (S_3 S_2 S_1, S_3(u))
     let  x = e_1 in  e_2
                 =  let  (S_1, τ_1)  = W(TE + {x : u}, e_1);
                         S_2         = Unify(S_1(u), τ_1);
                         (S_3, τ_2)  = W(S_2 S_1(TE) + {x : τ_1}, e_2);
                   in   (S_3 S_2 S_1, τ_2)
```

u's represent new type variables

Type Inference

```
Let fact = λn.if (n == 0) then 1
                 else n * fact (n-1)
In fact
```



Inferring Polymorphic Types

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