# The Hindley-Milner Type System
*(Continued)*

Arvind
Laboratory for Computer Science
M.I.T.

September 30, 2002

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

## Outline

- Hindley-Milner Type inference rules
- Type inference algorithm
- Overloading
- Type classes

## A mini Language
*to study Hindley-Milner Types*

*Expressions*

| | |
|---|---|
| E ::= c | constant |
| \| x | variable |
| \| $\lambda$x. 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.*

## A Formal Type System

*Types*

| | |
|---|---|
| $\tau$ ::= $\iota$ | base types |
| \| t | type variables |
| \| $\tau_1$--> $\tau_2$ | Function types |

*Type Schemes*

$\sigma$ ::= $\tau$
| $\forall$t. $\sigma_?$

*Type Environments*

TE ::= Identifiers --> 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 $\sigma$

## 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 Unify($\tau_1$, $\tau_2$) =
    case  ($\tau_1$, $\tau_2$) of
        ($\tau_1$, $t_2$) = [$\tau_1$ / $t_2$]
        ($t_1$, $\tau_2$) = [$\tau_2$ / $t_1$]
        ($\iota_1$, $\iota_2$) = 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
```

*Order in which sub-expressions are unified does not matter.*

## Instantiations

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

- Type scheme $\sigma$ can be *instantiated* into a type $\tau'$ by *substituting types for the bound variables of* $\sigma$, i.e.,

  $\tau' = S\ \tau$ for some S s.t. Dom(S) $\subseteq$ BV($\sigma$)

  - $\tau'$ is said to be an *instance of* $\sigma$ ($\sigma > \tau'$)
  - $\tau'$ is said to be a *generic instance of* $\sigma$ when S maps variables to new variables.

Example:

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

$t_3$ --> $t_2$ is a generic instance of $\sigma$
Int --> $t_2$ is a non generic instance of $\sigma$

## Generalization *aka Closing*

$$\mathrm{Gen}(TE,\tau) = \forall\, t_1 \ldots t_n.\ \tau \quad \text{where } \{ t_1 \ldots t_n \} = FV(\tau) - FV(TE)$$

- *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.

## Type Inference Rules

Typing: $\quad$ TE |-- e : $\tau$

Suppose we want to assert (prove) that give 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 $e_1$ has type $\tau$--> $\tau'$ and $e_2$ has the type $\tau$ .

## Type Inference Rules

Typing: $\quad$ TE |-- e : $\tau$

(App)
?

$$\frac{\text{TE |-- } e_1 : \tau\text{-->}\,\tau' \qquad \text{TE |-- } e_2 : \tau}{\text{TE |-- } (e_1\ e_2) : \tau'}$$

(Abs)

$$\frac{}{\text{TE |}\quad \lambda x.e : \tau \quad > \tau}$$

(Var)

$$\frac{}{\text{TE |}\quad x : \tau}$$

(Const)

$$\frac{}{\text{TE |}\quad c : \tau}$$

(Let)

$$\frac{}{\text{TE |}\quad (\textit{let}\ x = e_1\ \textit{in}\ e_2) : \tau}$$

## Generalization is restricted!

(Var) $$\frac{(x : \sigma)\ \varepsilon\ TE \qquad \sigma \geq \tau}{TE \vdash x : \tau}$$

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

Contrast:

(Gen) $$\frac{TE \vdash e : \tau \qquad t \notin FV(TE)}{TE \vdash e : \forall t.\tau}$$

(Spec) $$\frac{TE \vdash e : \forall t.\tau}{TE \vdash e : \tau\,[t'/t]}$$

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

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

## Soundness

- The proposed type system is *sound,* i.e. if $e : \tau$ then e indeed evaluates to a value in $\tau$.

- A method of proving soundness:
  - The semantics of the language is defined in terms of a value space that has integer values, Boolean values etc. as subspaces.
  - Any expression with a type error evaluates to a special value “wrong”.
  - There is no type expression that denotes the subspace “wrong”.

## Inference Algorithm

W(TE, e) returns $(S,\tau)$ such that S (TE) |-- e : $\tau$

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               =     ...
      (e_1 e_2)          =     ...
      let  x = e_1 in e_2  =     ...
```

## Inference Algorithm *(cont.)*

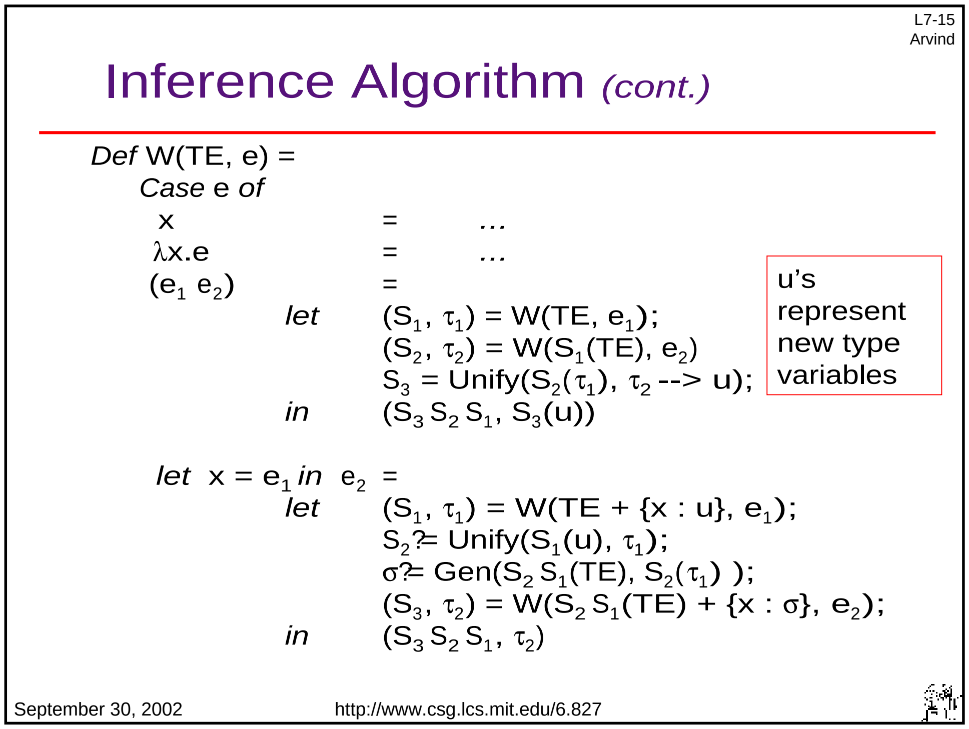
```
Def W(TE, e)  =
    Case e of
      x       =
        if  (x ∉ Dom(TE)) then Fail
                  else  let  ∀t_1...t_n.τ = TE(x);
                        in   ( { }, [u_i / t_i] τ)

    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  x = e_1 in  e_2
                  = ...
```

u's represent new type variables

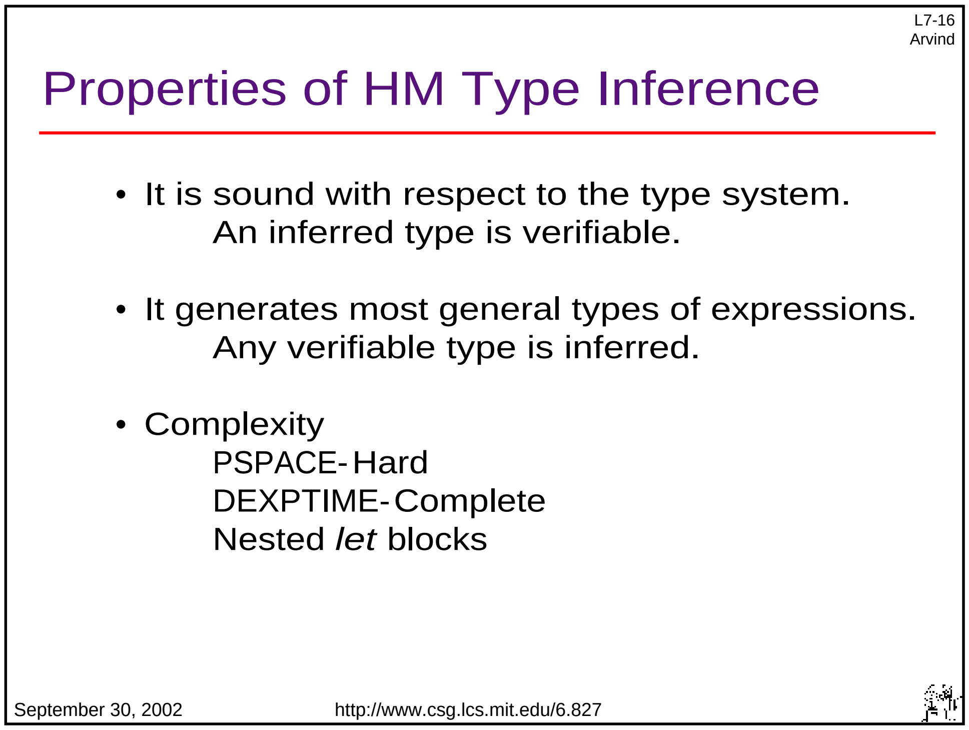


## Inference Algorithm *(cont.)*

```
Def W(TE, e) =
    Case e of
      x              =     …
      λx.e           =     …
      ($e_1$ $e_2$)        =
              let    ($S_1$, $\tau_1$) = W(TE, $e_1$);
                     ($S_2$, $\tau_2$) = W($S_1$(TE), $e_2$)
                     $S_3$ = Unify($S_2$($\tau_1$), $\tau_2$ --> u);
              in     ($S_3$ $S_2$ $S_1$, $S_3$(u))

      let x = $e_1$ in $e_2$ =
              let    ($S_1$, $\tau_1$) = W(TE + {x : u}, $e_1$);
                     $S_2$ ?= Unify($S_1$(u), $\tau_1$);
                     $\sigma$ ?= Gen($S_2$ $S_1$(TE), $S_2$($\tau_1$) );
                     ($S_3$, $\tau_2$) = W($S_2$ $S_1$(TE) + {x : $\sigma$}, $e_2$);
              in     ($S_3$ $S_2$ $S_1$, $\tau_2$)
```

u's represent new type variables



## Properties of HM Type Inference

- It is sound with respect to the type system.
  An inferred type is verifiable.
- It generates most general types of expressions.
  Any verifiable type is inferred.
- Complexity
  PSPACE-Hard
  DEXPTIME-Complete
  Nested *let* blocks

## Extensions

- Type Declarations
  Sanity check; can relax restrictions
- Incremental Type checking
  The whole program is not given at the same time, sound inferencing when types of some functions are not known
- Typing references to mutable objects
  Hindley-Milner system is unsound for a language with refs (mutable locations)
- Overloading Resolution

## Overloading *ad hoc polymorphism*

A symbol can represent multiple values each with a different type. For example:

+ represents

```
plusInt   :: Int -> Int -> Int
plusFloat :: Float -> Float -> Float
```

The *context* determines which value is denoted.

The overloading of an identifier is *resolved* when the unique value associated with the symbol in that context can be determined.

Compiler tries to resolve overloading but sometimes can't. The user must declare the type explicitly in such cases.

## Overloading vs. Polymorphism

Both allow a single identifier to be used for multiple types.

However, two concepts are very different:

1. All specific types of a *polymorphic* identifier are instances of a *most general* type.

2. A polymorphic identifier represents a *single function* semantically.

## The Most General Type

*The most general type* of twice is

```
∀t.(t -> t) -> (t -> t)
```

Any type can be substituted for **`t`** to get an instance of **`twice`**

```
(Int-> Int)       -> (Int -> Int)
(String -> String) -> (String -> String)
```

Overloaded + does not have a most general type.

An overloaded function may perform semantically unrelated operations in different contexts.

## Overloading in Haskell

Haskell has one of the most sophisticated overloading mechanism called *type classes*

*Type classes* allow overloading of user defined symbols

```
sqr x = x * x
```

Is the type of **sqr** **intSqr** or **FloatSqr** **?**

```
intSqr      :: Int -> Int
floatSqr    :: Float -> Float
```

In Haskell **sqr** can be overloaded and resolved based on its use.

## Type Classes
*making overloading less ad hoc*

Often a *collection of related functions* (e.g., +, -, *) need a common overloading mechanism and there is a *collection of types* (e.g., Int, Float) over which these functions need to be overloaded.

Type classes bring these two concepts together

```
class Num a where
   (==), (/=)        ::   a -> a -> Bool
   (+), (-), (*)     ::   a -> a -> a
   negate            ::   a -> a
   ...
instance Num Int where
    x == y           = integer_eq x y
    x + y            = integer_add x y
      …
instance Num Float where ...
```

## Overloaded Constants

`(Num t)` is read as a predicate
"`t` is an instance of class `Num`"

```
sqr  ::  (Num a) => a -> a
sqr x = x * x
```

What about constants? Consider

```
plus1 x = x + 1
```

If `1` is treated as an integer then `plus1` cannot be overloaded. In pH numeric literals are overloaded and considered a short hand for
`(fromInteger` *the_integer_1_value*`)`
where
```
fromInteger  ::  (Num a) => Integer -> a
```

## The Equality Operator

Equality is an overloaded and not a polymorphic function

```
class Eq a where
  (==)     ::   a -> a -> Bool
  (/=)     ::   a -> a -> Bool
```

Thus equality needs to be defined for each type of interest.

## Read and Show Functions

The raw input from a key board or output to the screen or file is usually a string. However, different programs interpret the string differently depending upon their type signature.

A program to calculate monthly mortgage payments may assign the following signatures:

```
read :: String -> Int     - principal, duration
read :: String -> Float   - rate
show :: Float  -> String  - monthly payments
```

what is the type of `read` and `show` ?

```
read :: String -> a
show :: a      -> String
```

*Polymorphic ?*

## Overloaded Read and Show

Haskell has a type class **Read** of "readable" types and a type class **Show** of "showable" types

```
read :: Read a => String -> a
show :: Show a => a      -> String
```

## Ambiguous Overloading

```
identity  :: String -> String
identity x = show (read x)
```

What is the type of `(read x)` ?

Cannot be resolved ! Many different types would do.

Compiler requires type declarations in such cases.

```
identity  :: String -> String
identity x = show ((read x) :: Int)
```

## Implementation

How does sqr find the correct function for * ?

```
sqr  ::  (Num a) => a -> a
sqr x = x * x
```

An overloaded function is compiled assuming an extra “dictionary” argument.

```
sqr’ = \class_inst x ->
                 (class_inst.(*)) x x
```

Then `(sqr 23)` will be compiled as

```
sqr’ IntClassInstance 23
```

Most dictionaries can be eliminated at compile time by function specialization.