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

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

Expressions

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

A Formal Type System

Types

\[
\tau ::= \iota \quad \text{base types} \\
    t \quad \text{type variables} \\
    \tau \rightarrow \tau_2 \quad \text{Function types}
\]

Type Schemes

\[
\sigma ::= \tau \\
    \forall t. \sigma
\]

Type Environments

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

Note, all the \(\forall\)'s occur in the beginning of a type scheme, i.e., a type \(\tau\) cannot contain a type scheme \(\sigma\)
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.

```python
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}$$\rightarrow$$\tau_{12}$, $\tau_{21}$$\rightarrow$$\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

- 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. $\text{Dom}(S) \subseteq \text{BV}(\sigma)$

  - $\tau'$ 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:

$\sigma = \forall t_1\ldots t_n\cdot \tau$

- $t_3$$\rightarrow$$t_2$ is a generic instance of $\sigma$
- $\text{Int}$$\rightarrow$$t_2$ is a non generic instance of $\sigma$
Generalization *aka* Closing

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

- **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.
Type Inference Rules

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

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

Then it must be the case that the same \( \text{TE} \) implies that \( e_1 \) has type \( \tau \rightarrow \tau' \) and \( e_2 \) has the type \( \tau \).

Type Inference Rules

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

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

(Abs) \[ \text{?} \]
\[ \text{TE} \vdash \lambda x. e : \tau \rightarrow \tau' \]

(Var) \[ \text{?} \]
\[ \text{TE} \vdash x : \tau \]

(Const) \[ \text{?} \]
\[ \text{TE} \vdash c : \tau \]

(Let) \[ \text{?} \]
\[ \text{TE} \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau \]
Generalization is restricted!

\begin{itemize}
\item (Var) \quad \frac{(x : \sigma) \in TE \quad \sigma \geq \tau}{TE \vdash x : \tau}
\item (Let) \quad \frac{TE + \{x : \tau\} \vdash e_1 : \tau \quad TE + \{x : \text{Gen}(TE, \tau)\} \vdash e_2 : \tau}{TE \vdash (let x = e_1 \ in e_2) : \tau}
\item (Gen) \quad \frac{TE \vdash e : \tau \quad t \notin \text{FV}(TE)}{TE \vdash e : \forall t. \tau}
\item (Spec) \quad \frac{TE \vdash e : \forall t. \tau}{TE \vdash e : \tau[t'/t]}
\item (Var) \quad \frac{(x : \tau) \in TE}{TE \vdash x : \tau}
\item (Let) \quad \frac{TE + \{x : \tau\} \vdash e_1 : \tau \quad TE + \{x : \tau\} \vdash e_2 : \tau}{TE \vdash (let x = e_1 \ in e_2) : \tau}
\end{itemize}

Contrast:

Soundness

\begin{itemize}
\item The proposed type system is sound, i.e. if $e : \tau$ then $e$ indeed evaluates to a value in $\tau$.
\item A method of proving soundness:
  \begin{itemize}
  \item The semantics of the language is defined in terms of a value space that has integer values, Boolean values etc. as subspaces.
  \item Any expression with a type error evaluates to a special value “wrong”.
  \item There is no type expression that denotes the subspace “wrong”.
  \end{itemize}
\end{itemize}
Inference Algorithm

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

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

```plaintext
Def \textit{W}(TE, e) =
Case \ e \ of
\ x = \ldots
\ \lambda x. e = \ldots
\ (e_1 e_2) = \ldots
\ \text{let} \ x = e_1 \ \text{in} \ e_2 = \ldots
```

Inference Algorithm (cont.)

```plaintext
Def \textit{W}(TE, e) =
Case \ e \ of
\ x = \ldots
\ if \ (x \not\in \text{Dom}(TE)) \ then \ \text{Fail}
\ else \ \text{let} \ \forall t_1, \ldots, t_n. \ \tau = TE(x);
\ in \ (\{\}, [u_i / t_i] \ \tau)
\lambda x. e = \ldots
\ \text{let} \ (S_1, \tau_1) = W(TE + \{ x : u \}, e);
\ in \ (S_1, S_1(u) \rightarrow \tau_1)
\ (e_1 e_2) = \ldots
\ \text{let} \ x = e_1 \ \text{in} \ e_2 = \ldots
```

u's represent new type variables
Inference Algorithm (cont.)

Def \( W(TE, e) = \)

Case \( e \) of

\( x \) = ... \\
\( \lambda x.e \) = ... \\
\( (e_1, e_2) \) =

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

let \( x = e_1 \) in \( e_2 = \)

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

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 \textit{ad hoc polymorphism}

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

+ represents

\begin{verbatim}
  plusInt :: Int -> Int -> Int
  plusFloat :: Float -> Float -> Float
\end{verbatim}

The \textit{context} determines which value is denoted.

The overloading of an identifier is \textit{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

$$\forall t. (t \rightarrow t) \rightarrow (t \rightarrow t)$$

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

$$\text{(Int} \rightarrow \text{Int}) \rightarrow \text{(Int} \rightarrow \text{Int})$$
$$\text{(String} \rightarrow \text{String}) \rightarrow \text{(String} \rightarrow \text{String})$$

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

\[ \text{sqr } x = x \times x \]

Is the type of \text{sqr} \text{IntSqr} or \text{FloatSqr}?

\[
\begin{align*}
\text{intSqr} & : \text{Int} \rightarrow \text{Int} \\
\text{floatSqr} & : \text{Float} \rightarrow \text{Float}
\end{align*}
\]

In Haskell \text{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

\[
\begin{align*}
\text{class } \text{Num } a \text{ where} \\
(==), (\neq) & :: a \rightarrow a \rightarrow \text{Bool} \\
(\), (-), (*) & :: a \rightarrow a \rightarrow a \\
\text{negate} & :: a \rightarrow a \\
\text{...}
\end{align*}
\]

\[
\begin{align*}
\text{instance } \text{Num } \text{Int } \text{where} \\
x == y & = \text{integer_eq} x y \\
x + y & = \text{integer_add} x y \\
\text{...}
\end{align*}
\]

\[
\begin{align*}
\text{instance } \text{Num } \text{Float } \text{where} \\
\text{...}
\end{align*}
\]
Overloaded Constants

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

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

What about constants? Consider

```haskell
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 shorthand for

```haskell
(fromInteger the_integer_1_value)
```

where

```haskell
fromInteger :: (Num a) => Integer -> a
```

The Equality Operator

Equality is an overloaded and not a polymorphic function

```haskell
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

\[
\text{identity} :: \text{String} \rightarrow \text{String} \\
\text{identity} \ x = \text{show} \ ((\text{read} \ x))
\]

What is the type of \((\text{read} \ x)\) ?

Cannot be resolved! Many different types would do.

Compiler requires type declarations in such cases.

\[
\text{identity} :: \text{String} \rightarrow \text{String} \\
\text{identity} \ x = \text{show} \ ((\text{read} \ x) :: \text{Int})
\]

Implementation

How does \text{sqr} find the correct function for * ?

\[
\text{sqr} :: (\text{Num} \ a) \Rightarrow a \rightarrow a \\
\text{sqr} \ x = x \ * \ x
\]

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

\[
\text{sqr'} = \text{\textbackslash class\_inst} \ x \rightarrow \\
\quad (\text{class\_inst}.(*)) \ x \ x
\]

Then \((\text{sqr} \ 23)\) will be compiled as

\[
\text{sqr'} \ \text{IntClassInstance} \ 23
\]

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