Hindley-Milner Type Inference

Outline of the Presentation :-

- Brief Historical Perspective of Type Theory
- Why do we need type inference
- Hindley-Milner Type system – what it offers

Hindley-Milner Type Inference

Historical Perspective :-

- Gottlob Frege - Begriffsschrift, a formal language, modeled upon arithmetic, for pure thought (1878).
- David Hilbert (1900)
- Prove consistency of arithmetic
- Axiomatize all of mathematics and prove its consistency.

Citation :- http://www.youtube.com/watch?v=h0OkptwfX4g and wikipedia

Hindley-Milner Type Inference

Historical Perspective :- continued...

- Bertard Russel (1908) - Russel's paradox
- “The set of all sets that do not contain themselves as members” - also barber problem.
- Introduced his version of Mathematical Logic based on type theory.
- Introduced hierarchy of Types
- “No totality can contain members defined in terms of itself”.

Citation :- http://www.youtube.com/watch?v=h0OkptwfX4g and wikipedia

Hindley-Milner Type Inference

Historical Perspective :- continued..

- Kurt Gödel (1931)
- Any sufficiently powerful system cannot be both consistent and complete.
- Arithmetic cannot prove its own consistency.
- Death knell for Hilbert's program.
- Type theory began as a failed program – couldn't prove consistency of arithmetic.

Citation :- http://www.youtube.com/watch?v=h0OkptwfX4g and wikipedia

Hindley-Milner Type Inference

Why do we need Type Inference :-

➔Readability – Difference in reading a C – Program and Python program. Important for maintenance of code.

➔Safety – Not all expressions in untyped languages makes sense. e.g. “**square root of banana”** or **“Navketan” + 5** .

➔Let us agree on the existence of bad expressions.

➔Any non-trivial choice of badness leads to an uncomputable subset, which compiler cannot distinguish good from bad.

Citation :- http://cs.au.dk/~mis/typeinf.pdf

Hindley-Milner Type Inference

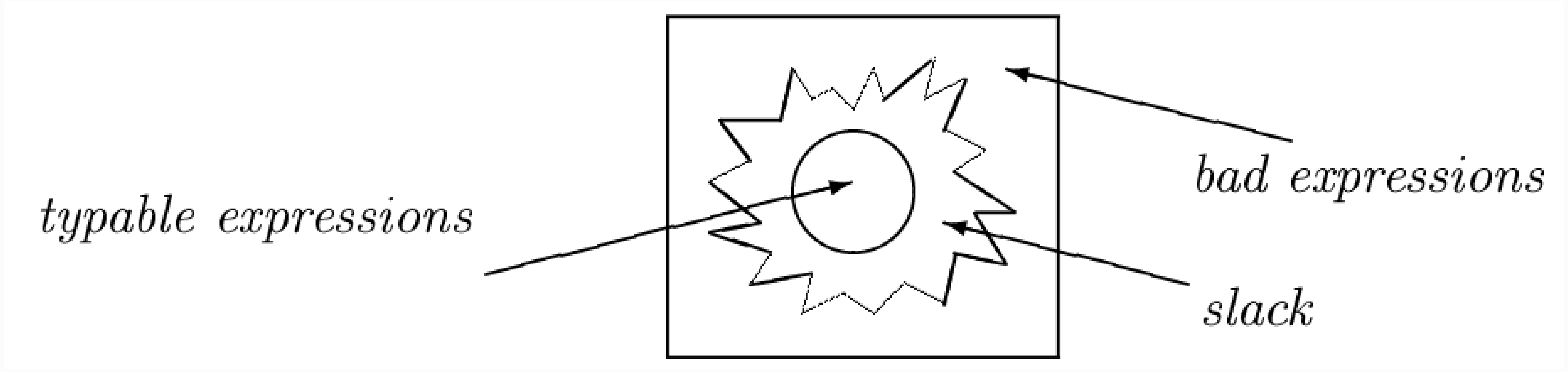
- **Lets look at an example for the above** :-
- **def foo(s: String) = s.length and**
- **def bar(x, y) = foo(x) + y**
- By looking at the definition of foo and bar we can easily put its type like **(String)=>Int** and **(String,Int)=>Int**.
- How ? Well **foo** expects a string as input parameter, therefore x parameter must be a string in **bar** as well (x passed to **foo**).
- Since **foo** returns length of the input string (integer), the parameter y must be of type integer (quite easy).
- Now lets modify this example a bit like, def bar(x,y) = foo(x) + “Navketan” .
- Compiler is going to flag this as error, since there is no overload version of '+' which combines an integer with string. As humans we can very easily determine this aspect, but for the machine (read compiler this job is not easy). We need to have some predefined rules which compiler can use and decide the legality of the program.
- Hindley-Milner or later Damas-Milner Type Inference algorithm does exactly this for us. It provides a certain rules which we can use to determine type and value of any legal expression . If program doesn't fit into any set of rules, its termed as illegal (error).

Citation :- http://www.codecommit.com/blog/scala/what-is-hindley-milner-and-why-is-it-cool

Hindley-Milner Type Inference

- **Why do we need further research in Type Inference** :-

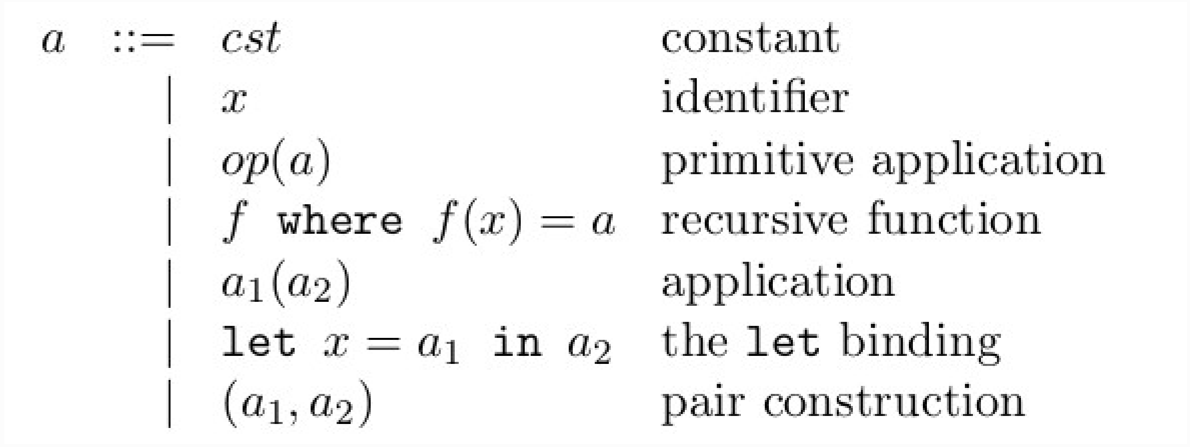
- To ensure safety of programs , we introduce a type system and some what as a compromise we have to sacrifice some good expressions. e.g.
- val x = 1;
- val x = 1 : int
- x + 2.3;
- Error-Can't unify int with real (Different type constructors) Found near +(x, 2.3). Static errors (pass2). **What we need to do is minimize this slack area.**



Citation :- http://cs.au.dk/~mis/typeinf.pdf

Hindley-Milner Type Inference

Hindley-Milner Syntax :- Expressions of the language are elements of term algebra generated using grammar below.



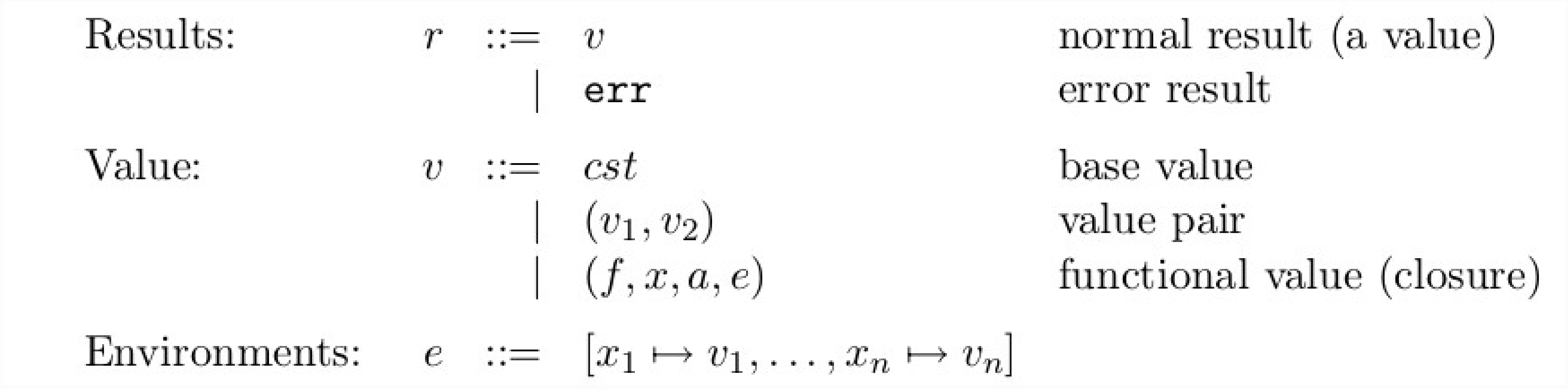
a	$::=$	cst	constant
	$\mid$	x	identifier
	$\mid$	$op(a)$	primitive application
	$\mid$	$f\ \texttt{where}\ f(x) = a$	recursive function
	$\mid$	$a_1(a_2)$	application
	$\mid$	$\texttt{let}\ x = a_1\ \texttt{in}\ a_2$	the `let` binding
	$\mid$	(a_1, a_2)	pair construction

Cst – constants , can be Integer,boolean,strings etc.
Identifiers usually have an infinite set. Domain of 'x' and 'f' .
'op' range over set of operators usually arithmetic, comparisons operator or if-then-else.
The let-construct defines a local scope and at the same time permits us to give a simple recursive definition.

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html

Hindley-Milner Type Inference

Hindley-Milner System :- Now we should write semantics to give meaning to the syntax above defined.
The evaluation argument e |- a => r should be read as “in the evaluation environment e, the expression a evaluates to the result r”
Semantic Objects:



Results:	r	$::=$	v	normal result (a value)
		$\mid$	$\texttt{err}$	error result
Value:	v	$::=$	cst	base value
		$\mid$	(v_1, v_2)	value pair
		$\mid$	(f, x, a, e)	functional value (closure)
Environments:	e	$::=$	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	

Hindley-Milner Type Inference

$$\text{Environments:} \quad e \quad ::= \quad [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$$

In environment expressions e, we assume the identifiers X_1....X_n to be distinct. Term e = [X_1 |→ V_1,X_2 |→ V_2,......X_n |→ V_n] is interpreted as impartial mapping with finite domain from identifiers to values i.e. mapping between X_i and V_i for all i = 1 to n. The empty mapping is written as [ ]. We also define extension of the environment e by V in X, written as e + x |→ v by

$$\begin{array}{rcl} [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] + x \mapsto v & = & [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, x \mapsto v] \\ & & \text{if } x \notin \{x_1, \ldots, x_n\} \\ [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] + x \mapsto v & = & [x_1 \mapsto v_1, \ldots, x_{i-1} \mapsto v_{i-1}, x \mapsto v, x_{i+1} \mapsto v_{i+1}, \ldots, x_n \mapsto v_n] \\ & & \text{if } x = x_i \end{array}$$

In other words,

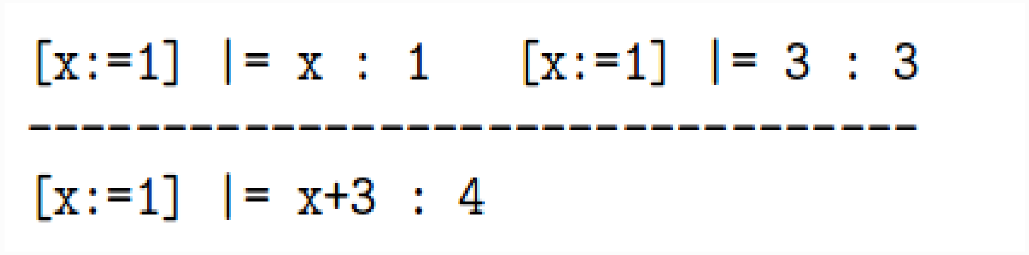
$$\begin{array}{rcl} \mathrm{Dom}(e + x \mapsto v) & = & \mathrm{Dom}(e) \cup \{x\} \\ (e + x \mapsto v)(x) & = & x \\ (e + x \mapsto v)(y) & = & e(y) \text{ for all } y \in \mathrm{Dom}(e),\ y \neq x. \end{array}$$

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html

Hindley-Milner Type Inference

Hindley-Milner System evaluation rules for values :- As an example look at the piece of code below.

- val x = 1;
- val x = 1 : int
- x + 3;
- val it = 4 : int



```
[x:=1] |= x : 1   [x:=1] |= 3 : 3
-------------------------------------
[x:=1] |= x+3 : 4
```

- x + d;
- Error-Value or constructor (d) has not been declared   Found near +(x, d)
- Static errors (pass2)

```
    [x := 1]  |= x : 1          [x := 1]  |= d : err (undeclared)
--------------------------------------------------------------------------
   [x := 1]  |= x + d : err
```

Hindley-Milner Type Inference

Hindley-Milner System evaluation rules for values :- Evaluation rules are defined as set of axioms and inference rules. Axiom P allows to conclude that preposition P holds if and only if P1,P2..Pn all other axioms evaluate to true . Here 'e' always means the current environment.

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{P}$$

Standard rules for evaluation of values are as follows :-

$$e \vdash cst \Rightarrow cst \qquad \frac{x \in \mathrm{Dom}(e)}{e \vdash x \Rightarrow e(x)} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x \Rightarrow \mathtt{err}}$$

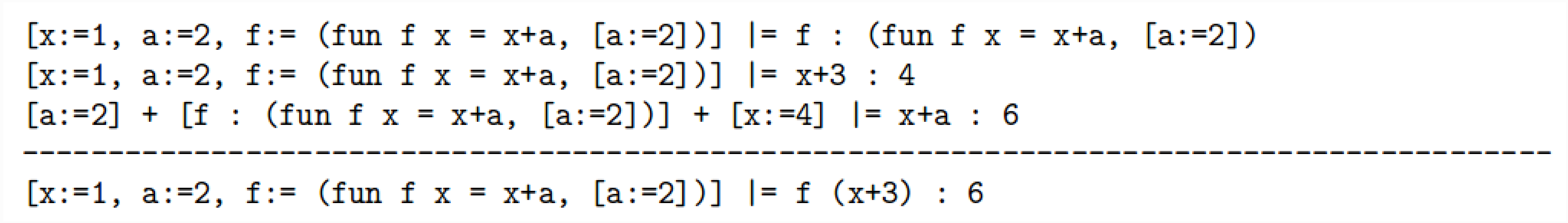
I) The left most rule says “Constant evaluates to itself”
II) The mid one evaluates an identifier = value bound to it in the environment.
III) The right most case of error arises in case of identifier doesn't belong to the domain of e

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html

Hindley-Milner Type Inference

Hindley-Milner System evaluation rules for values :- Another example .

val a = 2;
val a = 2 : int
fun f x = x + a;
val f = fn : int -> int
f 3;
val it = 5 : int
f (x+3);
val it = 6 : int



```
[x:=1, a:=2, f:= (fun f x = x+a, [a:=2])] |= f : (fun f x = x+a, [a:=2])
[x:=1, a:=2, f:= (fun f x = x+a, [a:=2])] |= x+3 : 4
[a:=2] + [f : (fun f x = x+a, [a:=2])] + [x:=4] |= x+a : 6
------------------------------------------------------------------------------
[x:=1, a:=2, f:= (fun f x = x+a, [a:=2])] |= f (x+3) : 6
```

Hindley-Milner Type Inference

Hindley-Milner System evaluation rules for values :- Another example .

f (x+d);
Error-Value or constructor (d) has not been declared   Found near f(+(x, d))
Static errors (pass2)

```
[x:=1, a:=2, f:= (fun f x = x+a, [a:=2])] |= f : (fun f x = x+a, [a:=2])
[x:=1, a:=2] |= d : err
[x:=1, a:=2, f:= (fun f x = x+a, [a:=2])] |= x+d : err
[a:=2] + [f : (fun f x = x+a, [a:=2])] + [x:=x+d] |= x+a : err
---------------------------------------------------------------------------------
[x:=1, a:=2, f:= (fun f x = x+a, [a:=2])] |= f (x+d) : err
```

Hindley-Milner Type Inference

- Standard rules for evaluation of values are as follows :-

$$e \vdash (f \ \texttt{where} \ f(x) = a) \Rightarrow (f, x, a, e)$$

A function evaluates to a closure : an object that combines the unevaluated body of a function (f,x,a) with environment e at the time of function definition.

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e \vdash a_2 \Rightarrow v_2}{e \vdash (a_1, a_2) \Rightarrow (v_1, v_2)} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}}$$

Pairs normally evaluate to another pair. If one of them is not available in the environment or causes an error, then error is raised. That way evaluation becomes somewhat sequential (left-to-right).

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html

Hindley-Milner Type Inference

- Standard rules for evaluation of values are as follows :-

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e + x \mapsto v_1 \vdash a_2 \Rightarrow r_2}{e \vdash \texttt{let } x = a_1 \texttt{ in } a_2 \Rightarrow r_2} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash \texttt{let } x = a_1 \texttt{ in } a_2 \Rightarrow \texttt{err}}$$

let binding evaluates its 1st argument and associates the obtained value to the bounded identifier, enriches the environment and then evaluates the second argument in the enriched environment, which will be result of the whole let expression. As in the earlier case, when one argument determination raises an error, its an error overall.

$$\frac{e \vdash a_1 \Rightarrow (f, x, a_0, e_0) \qquad e \vdash a_2 \Rightarrow v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \Rightarrow r_0}{e \vdash a_1(a_2) \Rightarrow r_0}$$

Expression a1 should evaluate to the closure (f,x,a0,e0) . a2 evaluate to v2. We then evaluate the function body a0 from the closure in the environment e0 and also enriching e0 by two more bindings , argument x and function name f.

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html

Hindley-Milner Type Inference

- Standard rules for evaluation of values are as follows :-

$$\frac{e \vdash a_1 \Rightarrow r_1 \qquad r_1 \text{ does not match } (f, x, a_0, e_0)}{e \vdash a_1(a_2) \Rightarrow \mathtt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \mathtt{err}}{e \vdash a_1(a_2) \Rightarrow \mathtt{err}}$$

Two possible error cases are listed here. Clearly if a1 is evaluated to anything other than a function closure, will be erroneous scenario. Second case if the evaluation of argument a2 runs into error.

$$\frac{e \vdash a_1 \Rightarrow \mathtt{true} \qquad e \vdash a_2 \Rightarrow r_2}{e \vdash \mathtt{if}\ a_1\ \mathtt{then}\ a_2\ \mathtt{else}\ a_3 \Rightarrow r_2} \qquad \frac{e \vdash a_1 \Rightarrow \mathtt{false} \qquad e \vdash a_3 \Rightarrow r_3}{e \vdash \mathtt{if}\ a_1\ \mathtt{then}\ a_2\ \mathtt{else}\ a_3 \Rightarrow r_3}$$

$$\frac{e \vdash a_1 \Rightarrow r_1 \qquad r_1 \notin \mathtt{Bool}}{e \vdash \mathtt{if}\ a_1\ \mathtt{then}\ a_2\ \mathtt{else}\ a_3 \Rightarrow \mathtt{err}}$$

This if-then-else rule is slightly different from the rest in the sense that 1st argument a1 always evaluate to bool value (if condition).

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html

Hindley-Milner Type Inference

- **Typing** : -Typing rules associate types (type expressions) to expressions, just like evaluation results to expressions. The main feature of Damas-Milner type inference system is polymorphism, i.e. same expression can evaluate to multiple types. We define a simple type system as follows :-

$$\begin{array}{rcllll} \iota & \in & \texttt{TypBas} & = & \{\texttt{int};\texttt{bool};\ldots\} & \text{base types} \\ \alpha,\beta,\gamma & \in & \texttt{VarTyp} & & & \text{type variables} \end{array}$$

Type of type expressions :-

$$\begin{array}{rcll} \tau & ::= & \iota & \text{base type} \\ & \mid & \alpha & \text{type variable} \\ & \mid & \tau_1 \rightarrow \tau_2 & \text{function type} \\ & \mid & \tau_1 \times \tau_2 & \text{product type} \end{array}$$

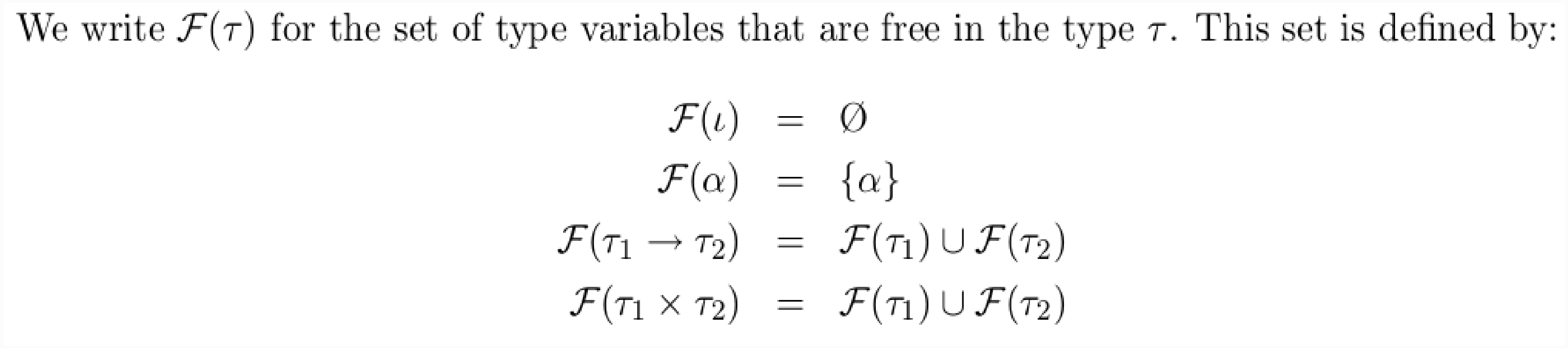
Example of type variable :- Vector in C++.

Std::vector<T> m_demoVector; // here T is a type variable

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html

Hindley-Milner Type Inference

- **Free Variables** :- Set of variables without quantification (w.r.t certain formula).



We write $\mathcal{F}(\tau)$ for the set of type variables that are free in the type τ. This set is defined by:

$$\begin{array}{rcl} \mathcal{F}(\iota) & = & \emptyset \\ \mathcal{F}(\alpha) & = & \{\alpha\} \\ \mathcal{F}(\tau_1 \rightarrow \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) \\ \mathcal{F}(\tau_1 \times \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) \end{array}$$

Example of free variable :-

Ay Ez P(x,y,z) , where P is a predicate . Here x is a free variable w.r.t given formula.
Also note that free variable is defined only for type variables and not base types.

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html

Hindley-Milner Type Inference

Substitution :- Substitutions are finite mappings from type variables to type expressions. A substitution instance of a propositional formula is a second formula obtained by replacing symbols of the original formula by other formulas. e.g. (R → S) & (T → S) is a substitution of P & Q .

A substitution φ naturally extends to an homomorphism of type expressions, written $\overline{\varphi}$, and defined by:

$$\begin{aligned}
\overline{\varphi}(\iota) &= \iota \\
\overline{\varphi}(\alpha) &= \varphi(\alpha) \text{ if } \alpha \in \mathrm{Dom}(\varphi) \\
\overline{\varphi}(\alpha) &= \alpha \text{ if } \alpha \notin \mathrm{Dom}(\varphi) \\
\overline{\varphi}(\tau_1 \to \tau_2) &= \overline{\varphi}(\tau_1) \to \overline{\varphi}(\tau_2) \\
\overline{\varphi}(\tau_1 \times \tau_2) &= \overline{\varphi}(\tau_1) \times \overline{\varphi}(\tau_2)
\end{aligned}$$

Proposition :-

(Substitution and free variables) *For all types τ and all substitutions φ, we*

$$\mathcal{F}(\varphi(\tau)) = \bigcup_{\alpha \in \mathcal{F}(\tau)} \mathcal{F}(\varphi(\alpha))$$

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html and http://en.wikipedia.org/wiki/Substitution_(logic)

Hindley-Milner Type Inference

Type Schemes :- Type schemes are basically set of types which can be obtained by substituting types for variables in a consistent manner. Denoted usually by letter σ and given by the following grammar :-

$$\sigma ::= \forall \alpha_1 \ldots \alpha_n . \tau \qquad \text{type scheme}$$

Quantified variables are treated as set of distinct variables. Their ordering is insignificant. We distinguish between type schemes only by renaming a bounded variable or by introduction or suppression of quantified variables.

$$\begin{aligned} \forall \alpha_1 \ldots \alpha_n . \tau &= \forall \beta_1 \ldots \beta_n . [\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau) \\ \forall \alpha \alpha_1 \ldots \alpha_n . \tau &= \forall \alpha_1 \ldots \alpha_n . \tau \quad \text{if } \alpha \notin \mathcal{F}(\tau) \end{aligned}$$

Free Variables in the type scheme is given as :-

$$\mathcal{F}(\forall \alpha_1 \ldots \alpha_n . \tau) = \mathcal{F}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\}.$$

Substitutions to the type scheme is given as :-

$$\varphi(\forall \alpha_1 \ldots \alpha_n . \tau) = \forall \alpha_1 \ldots \alpha_n . \varphi(\tau)$$

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html and http://cs.au.dk/~mis/typeinf.pdf

Hindley-Milner Type Inference

Type Schemes :- Example

scheme $(\alpha \to \beta) \to \mathsf{list}(\alpha)$ defines the following infinite set:

$$\begin{array}{rcl}
(\mathsf{Int}\to\mathsf{Int}) & \to & \mathsf{list}(\mathsf{Int}) \\
(\mathsf{Bool}\to\mathsf{Int}) & \to & \mathsf{list}(\mathsf{Bool}) \\
(\mathsf{Bool}\to\mathsf{Bool}) & \to & \mathsf{list}(\mathsf{Bool}) \\
(\mathsf{Int}\to\mathsf{Bool}) & \to & \mathsf{list}(\mathsf{Int}) \\
 & \vdots & \\
((\mathsf{Int}\to\mathsf{Bool})\to\mathsf{list}(\mathsf{Bool})) & \to & \mathsf{list}(\mathsf{Int}\to\mathsf{Bool}) \\
 & \vdots &
\end{array}$$

Only precondition being all occurrences of any type variable should be replaced all together with the same type.

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html and http://cs.au.dk/~mis/typeinf.pdf

Hindley-Milner Type Inference

Typing environments :- A typing environment, 'E', is a finite mapping between from identifiers to type Schemes. We can think of 'E' as a symbol table providing a mapping from identifiers to types and table gets updated with each new declaration.

$$E \quad ::= \quad [x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]$$

The image of an environment E by a substitution is defined as :-

$$\varphi([x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]) = [x_1 \mapsto \varphi(\sigma_1), \ldots, x_n \mapsto \varphi(\sigma_n)].$$

Typing Rules :- typing rules for the language are very similar to the evaluation rules for the expressions. The typing rule ' E |- e : t ' is read as “ in typing environment E expression e has type t”. The environment E associates a type to each identifier that appear in the expression e.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

An identifier x (in typing environment E) in an expression can be given any type that is an instance of the $\tau \leq \sigma$heme associated with the identifier . Type τ is an instance of type scheme $\sigma = \forall \alpha_1 \ldots \alpha_n . \tau_0$ and denoted by $\tau \leq \sigma$ if and only if $\tau = \varphi(\tau_0)$, where φ s a substitution function with domain $\{\alpha_1 \ldots \alpha_n\}$.

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html and http://cs.au.dk/~mis/typeinf.pdf

Hindley-Milner Type Inference

Typing environments :- Example
Lets look at the old sample program

```
val x = 1;
val x = 1 : int
val a = 2;
val a = 2 : int
fun f x = x + a;
val f = fn : int -> int
```

- E |= x : Int   E |= a : Int   E |= + : Int → Int → Int   E |= x + a : Int
- E + [f : (fun f x = x+a : Int → Int] |= x+a : Int
- --
- E |= f:= (fun f x = x+a) : Int → Int

Hindley-Milner Type Inference

Typing Rules :- A function definition has type $\tau_1 \rightarrow \tau_2$ with its formal parameter x has type τ_1 and function body has type τ_2 . It's internal name 'f' has the same type $\tau_1 \rightarrow \tau_2$

$$\frac{E + f \mapsto (\tau_1 \rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \rightarrow \tau_2}$$

Function application of the form a1(a2) has type correspondence between arguments and function parameter. Type of function result is the type of whole application node. The rule is given below as

$$\frac{E \vdash a_1 : \tau_2 \rightarrow \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

The let construct introduces polymorphic types in the type environment. This is due to the generalization operator Gen, which builds a type scheme from a type and environment.

$$\texttt{Gen}(\tau, E) = \forall \alpha_1 \ldots \alpha_n . \tau \quad \text{where} \quad \{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{F}(E).$$

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let} \ x = a_1 \ \texttt{in} \ a_2 : \tau_2}$$

The typing environment 'E' is enhanced by the declaration of let construct.

Citation :- http://pauillac.inria.fr/~xleroy/bibrefs/Leroy-thesis.html and http://cs.au.dk/~mis/typeinf.pdf

Hindley-Milner Type Inference

Typing environments :- Example
Lets look at the another more complicated example

```
fun succ(y) = y+1;
val succ = fn : int -> int
map succ([1,2,3]);
val it = [2, 3, 4] : int list
```

- E |= 1 : Int E |= y : Int   E |= + : Int → Int → Int
- E |= y + 1 : Int
- E + [f : (fun succ(y) = y+1 : Int → Int] |= e : Int
- E + [map succ([1,2,3]) : list(Int) → list(Int)] |= list(Int)
- --
- E |= map succ([1,2,3]) : list(Int)

Hindley-Milner Type Inference

Typing Rules :- The typing rules for constant, operator overloading and pair expressions are below.

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \qquad \frac{\tau \leq \mathtt{TypCst}(cst)}{E \vdash cst : \tau} \qquad \frac{\tau_1 \rightarrow \tau_2 \leq \mathtt{TypOp}(op) \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

$$\begin{array}{rcll}
\mathtt{TypCst}(i) & = & \mathtt{int} & (i = 0, 1, \ldots) \\
\mathtt{TypCst}(b) & = & \mathtt{bool} & (b = \mathtt{true} \text{ or } \mathtt{false}) \\
\mathtt{TypOp}(+) & = & \mathtt{int} \times \mathtt{int} \rightarrow \mathtt{int} & \\
\mathtt{TypOp}(\mathtt{ifthenelse}) & = & \forall \alpha.\ \mathtt{bool} \times \alpha \times \alpha \rightarrow \alpha &
\end{array}$$

Hindley-Milner Type Inference

- Conclusion

- Q & A