Why Type Checking and Type Inference

Main reasons for introducing type checking or type inferencing in programming languages was to ensure safety and readability of programs.

First look at easier one, readability. Consider reading and maintaining of a simple program written in C and in python. One can easily distinguish that reading C code is faster and relatively easier than python. Reason, presence of predefined data types in C. The reader/programmer has a rough idea of what sort of values it should expect for a particular identifier (because of the presence of data types).

Second reason is safety of the program. Programs in any programming langauge are made up of a series of statements. But all statements of the program doesnot makes sense. e.g. i = 5 + "Navketan" or "Banana" * "Orange". We need to agree that some statements doest not makes sense or they are bad. As a programmer one needs some help from compiler to determine legality of program statements. For the compiler to make such a judgement , some set of rules is required through which it can determine whether statements are good or bad. Hindley-Milner(HM) type inferencing provides just that. It provides a set rules for inferencing type (and value) of a particular expression by the compiler (or interpreter). If its able to find a unique type for the current program statement(or expression) it marks the statement as correct. If the statement doesn't follow its set of rules or compiler is not able to find a unique type for a particular statement then it flags an error. In the following sections we are going to present the basic language construct(Since the full ML language is too complex) and the set of rules that HM system offers and give examples of how HM type system actually works.

Syntax of Hindley-Milner type system

Expressions of the language are elements of term algebra generated using grammar below :

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

x and f range over infinite set of identifiers. cst are constants which can be Integers, boolean, character strings etc. op ranges over operators like arithmetic operator, comparasions or projection. conditional constructs like if...then...else are also treated as operators. The expression f where $f(x) = a$ defines function

whose parameter is x and result a. f is the internal name of the function. Inside a, f is considered bound to the function being defined. The let construct defines a local scope and at the same time permits us to give simple recursive definition.

Example: Here is an example program for factorial calculation.

fun fact n = let fun f 0 = 1 | f n = n * f (n - 1) in if (n < 0) then raise Fail "negative argument" else f n end;

Semantics of Hindley-Milner type system

The evaluation argument $e \vdash a \Rightarrow r$ 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)
	$\mid err$	error result
Value:	$v ::= cst$	base value
	$\mid (v_1,v_1)$	value pair
	$\mid (f,x,a,e)$	functional value(closure)
Environments:	$e ::= [x_1 \mapsto v_1,x_2 \mapsto v_2,....,x_n \mapsto v_n]$	

In environment expressions e, we assume the identifiers $x_1....x_n$ to be distinct. Term $e = [x_1 \mapsto v_1,x_2 \mapsto v_2,....,x_n \mapsto v_n]$ is interpreted as impartial mapping with finite domain from identifiers to values i.e. mapping between x_1 and v_1 $\forall$ i = 1 to n. The empty mapping is written as []. We also define extension of the environment e by v in x, written as $e + x \mapsto v$ by :

$[x_1 \mapsto v_1,....,x_n \mapsto v_n] + x \mapsto v = [x_1 \mapsto v_1,....,x_n \mapsto v_n,x \mapsto v\ ]$ if $x \notin \{\ x_1,x_2,.....x_n\ \}$

$[x_1 \mapsto v_1,....,x_n \mapsto v_n] + x \mapsto v = [x_1 \mapsto v_1,....x_{i-1} \mapsto v_{i-1},x_i \mapsto v_i,x_{i+1} \mapsto v_{i+1},...,x_n \mapsto v_n\ ]$ if $x = x_i$

In other words,

$\mathrm{Dom}(e + x \mapsto v) = \mathrm{Dom}(e) \bigcup \{x\}$

$(e + x \mapsto v)(x) = x$

$(e + x \mapsto v)(y) = e(y)\ \forall y \in \mathrm{Dom}(e),\ y \neq x$

Evaluation rule

Evaluation rules are set up so that we are able to define the judgement $e \vdash a \Rightarrow r$. Defined as set of axioms and inference rules. Axiom P allows to conclude that preposition P holds if and only if $P_1,P_2..P_n$ all other axioms evaluate to true . Here 'e' always means the current environment. And it's denoted as :

$$\frac{P_1\ P_2\ ....P_n}{P}$$

The Standard rules of evaluation are :

$$\frac{}{e \vdash a \Rightarrow r} \qquad \frac{x \in Dom(e)}{e \vdash a \Rightarrow r} \qquad \frac{x \notin Dom(e)}{e \vdash a \Rightarrow err}$$

These rules states that a constant evaluates to itself, identifier evaluates to the corresponding value bound to it in the environment and if the identifier is not present in the environment then it raises an error.

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

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.

$$\frac{e_1 \vdash a_1 \Rightarrow v_1 \quad e_2 \vdash a_2 \Rightarrow v_2}{e \vdash (a_1, a_2) \Rightarrow (v_1, v_2)} \qquad \frac{e_1 \vdash a_1 \Rightarrow err}{e \vdash (a_1, a_2) \Rightarrow err} \qquad \frac{e_2 \vdash a_2 \Rightarrow err}{e \vdash (a_1, a_2) \Rightarrow err}$$

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).

$$\frac{e_1 \vdash a_1 \Rightarrow v_1 \quad e + x \mapsto v_1 \vdash a_2 \Rightarrow r_2}{e \vdash let \;\; x = a_1 \;\; in \;\; a_2 \;\; \Rightarrow r_2} \qquad \frac{e_1 \vdash a_1 \Rightarrow err}{e \vdash let \; x = a_1 \;\; in \;\; a_2 \;\; \Rightarrow err}$$

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.

$$\frac{e_1 \vdash a_1 \Rightarrow (f, x, a_0, e_0) \quad e \vdash a_2 \Rightarrow v_2 \;\; e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \Rightarrow r_0}{e \vdash a_1(a_2) \;\; \Rightarrow r_2}$$

Expression a_1 should evaluate to the closure (f,x,a_0,e_0) . a_2 evaluate to v_2. We then evaluate the function body a_0 from the closure in the environment e_0 and also enriching e_0 by two more bindings , argument x and function name f.

$$\frac{e_1 \vdash a_1 \Rightarrow r_1 \; r_1 \; does \; not \; match \; (f, x, a_0, e_0)}{e \vdash a_1(a_2) \;\; \Rightarrow err} \qquad \frac{e_2 \vdash a_2 \Rightarrow err}{e \vdash a_1(a_2) \;\; \Rightarrow err}$$

Two possible error cases are listed here. Clearly if a_1 is evaluated to anything other than a function closure, will be erroneous scenario. Second case if the evaluation of argument a_2 runs into error.

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

$$\frac{e \vdash a \Rightarrow r_1 \quad r_1 \notin Bool}{e \vdash if \;\; a_1 \;\; then \;\; a_2 \;\; else \;\; a_3 \;\; \Rightarrow err}$$

This if-then-else rule is slightly different from the rest in the sense that 1st

argument a_1 always evaluate to bool value (if condition).

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 :-

$$\begin{array}{rll} \iota \in \text{TypBas} = \{int; bool; ....\} & & \text{base types} \\ \alpha, \beta, \gamma \in \text{TypVar} & & \text{type variables} \end{array}$$

Set of type expressions :-

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

Free Variables :- Set of variables without quantification (w.r.t certain formula).We write $F(\tau)$ for the set of type variables that are free in the type τ . This set is defined by:

$$\begin{array}{l} F(\tau) := \phi \\ F(\alpha) := \{\alpha\} \\ F(\tau_1 \rightarrow \tau_2) := F(\tau_1) \bigcup F(\tau_2) \\ F(\tau_1 \times \tau_2) := F(\tau_1) \bigcup F(\tau_2) \end{array}$$

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

$$\begin{array}{l} \overline{\varphi}(\tau) := \iota \\ \overline{\varphi}(\alpha) := \varphi(\alpha) \; if \; \alpha \in Dom(\varphi) \\ \overline{\varphi}(\alpha) := \alpha \; if \; \alpha \notin Dom(\varphi) \\ \overline{\varphi}(\tau_1 \rightarrow \tau_2) := \overline{\varphi}(\tau_1) \rightarrow \overline{\varphi}(\tau_2) \\ \overline{\varphi}(\tau_1 \times \tau_2) := \overline{\varphi}(\tau_1) \times \overline{\varphi}(\tau_2) \end{array}$$

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 :-

$$\sigma ::= \forall\alpha_1\alpha_2....\alpha_n.\tau \qquad \text{type scheme}$$

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.

$$\forall\alpha_1\alpha_2....\alpha_n.\tau = \forall\beta_1\beta_2....\beta_n.[\alpha_1 \mapsto \beta_1, ...\alpha_n \mapsto \beta_n]\tau$$
$$\forall\alpha\alpha_1\alpha_2....\alpha_n.\tau = \forall\forall\alpha_1\alpha_2....\alpha_n.\tau \;\; if\ \alpha \notin F(\tau)$$

Free Variables in the type scheme is given as :-

$$F(\forall\alpha_1\alpha_2....\alpha_n) = F(\tau) \setminus \{\alpha_1, \alpha_2, ...\alpha_n\}$$

Substitutions to the type scheme is given as :-

$$\varphi(\forall\alpha_1\alpha_2....\alpha_n.\tau) = \forall\alpha_1\alpha_2....\alpha_n.\varphi(\tau)$$

Type Scheme Example : scheme$(\alpha \to \beta) \to$ list(α) defines the following infinite set :

(Int→Int)→list(Int)
(Bool→Int)→list(Bool)
(Bool→Bool)→list(Bool)
(Int→Bool)→list(Int) ....
((Int→Bool)→Int)→list((Int→Bool))

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

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.

$$E ::= [x_1 \mapsto v_1, x_2 \mapsto v_2...x_n \mapsto v_n]$$

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

$$\varphi([x_1 \mapsto v_1, x_2 \mapsto v_2...x_n \mapsto v_n]) = [x_1 \mapsto \varphi(v_1), x_2 \mapsto \varphi(v_2)...x_n \mapsto \varphi(v_n)]$$

Typing Rules: Typing rules for the language are very similar to the evaluation rules for the expressions. The typing rule $E \vdash e : t$ is read as "in typing environment E expression e has type t".$E \mid> p : E1$ is read as "in type environment E, typing program p results in type environment E1 ".The environment E associates a type to each identifier that appear in the expression e.

Type inferencing rules : below we are writing the set of rules which we use for type inferencing.

$$\frac{E(b) = t}{E \vdash b : t} \qquad \frac{b \in \{true, false\}}{E \vdash b : bool} \qquad \frac{E \vdash b : t}{E \vdash (b) : t}$$

$$\frac{E \vdash o : t1 \to t2 \quad E \vdash b : t1}{E \vdash o\ b : t2} \qquad \frac{E \vdash o : t1 \to t2 \to t \quad E \vdash b1 : t1 \quad E \vdash b2 : t2}{E \vdash b1\ o\ b2 : t}$$

$$\frac{E \vdash b1 : bool \quad E \vdash b2 : t \quad E \vdash b3 : t}{E \vdash if\ b1\ then\ b2\ else\ b3 : t}$$

$$\frac{E \vdash b1 : t2 \to t1 \quad E \vdash b2 : t1}{E \vdash b1\ b2 : t1} \qquad \frac{E \mid> p : E1 \quad E1 \vdash e : t}{E \vdash let\ p\ in\ e\ end : t}$$

$$\frac{E \vdash e : t \quad E + [b := t] \vdash e : t}{E \mid> val\ b\ =\ e\ : E + [b := t]} \qquad \frac{E + [f := t1 \to t2] + [b := t] \vdash e : t2}{E \mid> fun\ f\ b\ =\ e\ : E + [f := t1 \to t2]}$$

Corrected Example on type evaluation on map (ref slide :)

Lets look at the example on map :
fun succ(y) = y+1;
val succ = fn : int → int
map succ([1,2,3]);
val it = [2, 3, 4] : int list

The function map takes two input parameters. As we see, the second argument of map function is a list and its first argument is a function which acts on list items. Here we are using the in-built map function, but we need body of the function and based on the body we can actually compute the type of the function. So i am writing a sample implementation of map function.

fun map f [ ] = [ ] | map f (h::t) = (f h)::(map f t);

Now looking at the function body, first part takes an empty list and returns an empty list (this doesn't give us any information). In the second part of the definition (after |) gives us enough information for the generation of constraints.

1) Type of function map is $map : T_1 * T_2 \to T_3$ where T_1, T_2 and T_3 are some type variables,that most generic definition of a function that takes two arguments and returns one. Also since the function f takes it argument from the second argument of the map function so $map : T_1 \to T_2 \to T_3$ where T_1
2) Let's take a generic type definition for function f (1st argument of the map function). It can noted from the above definition that it takes one argument and returns one as well. so $f : T_4 \to T_5$. Since the first argument of the map function is this function f, therefore $f : T_4 \to T_5 = f : T_1$, which is possible if and only if $T_4 \to T_5 = T_1$
3) Let h has some type T_6 denoted as $h : T_6$.
4) Let t has some type T_7 denoted as $t : T_7$. t has the type of list of type h [going by the pattern h::t]. $T_7 = \text{list}(T_6)$.
5) Going by the pattern of the function body, function f takes h as it's argu-

ment [(f h)::(map f t)] . This is possible if and only if $T_4 = T_6$.
6) Going by point 3 and 4, $T_7 = \text{list}(T_6) = \text{list}(T_4)$.
7) Going by the pattern of the function body, function map returns a list of (f h) type of elements. from point 2, $(fh)\colon T_5$. $T_3 = \text{list}(T_5)$.
8) Putting it all together,
$map\colon T_1 * T_2 \to T_3 = map\colon (T_4 \to T_5) \to list(T_4) \to list(T_5)$.
9) Now replace values of T_4 and T_5 on all places in the type definition of map, we get map : ('a→'b)→ 'a list → 'b list.

$E \vdash 1\colon Int \quad E \vdash +\colon Int \to Int \to Int \quad E \vdash y\colon Int \quad E \vdash y+1\colon Int$

$E \vdash [1,2,3]\colon list(Int) \qquad E + 'alist = list(Int) \vdash 'a\colon Int$ [Comment :- Since input is a list of Int so from point 9 we can conclude that input to the function f(succ in our case is type Int)]

$E + [f\colon (fun\ succ(y) = y+1)\colon Int \to Int] \vdash e\colon Int$ [Comment :- Since input to the function f(succ) is Int, we are computing that return type of f(succ) : 'b = Int]

$E \vdash 'b\colon Int \quad E \vdash 'blist : list(Int)$ [Commment :- return type of map is 'b list = list(Int)]

$$\frac{E + [f\colon (fun\ succ(y) = y+1)\colon Int \to Int] + [map\colon (Int \to Int) \to list(Int) \to list(Int)] \vdash list(Int)}{E \vdash map\ succ([1,2,3])\colon list(Int)}$$

Conclusion :

Well basically we have seen that type inferencing and expression evaluation revolves around a fixed set of rules. This is something that makes Hindley-Milner(HM) type inferencing algorithm relatively faster. HM system not only offers type inferencing system but it inherently does type checking for us as well(ref error example on the slides). One of the most important aspects of HM type system is parametric polymorphism (depicted by above example on map). HM algorithm picks the most general type from the available type scheme for a particular type variable.

References :

1. Phd Thesis - Xavior Leroy - $http://pauillac.inria.fr/\ xleroy/bibrefs/Leroy-these.html$
2. Poly Morphic Type Inference - $http://cs.au.dk/\ mis/typeinf.pdf$
3. $http://en.wikipedia.org/wiki/Type_inference$ - For Definition