# TYPE INFERENCE IN PROLOG AND ITS APPLICATION

Tadashi KANAMORI, Kenji HORIUCHI

Central Research Laboratory
Mitsubishi Electric Corporation
Tsukaguchi-Honmachi 8-I-I,Amagasaki,Hyogo,Japan 661

Abstract

In this paper we present a type inference method for Prolog programs. The new idea is to describe a superset of the success set by associating a type substitution (an assignment of sets of ground terms to variables) with each head of definite clause. This approach not only conforms to the style of definition inherent to Prolog but also gives some accuracy to the types infered. We show the basic computation method of the superset by sequential approximation as well as an incremental method to utilize already obtained results. We also show its application to verification of Prolog programs.

## 1. Introduction

It gives usefull information not only to programmers but also to met a-processing systems to infer characteristics of execution-time behavior from program texts. Especially in Prolog, it is usefull to know of what type a variable in a query is when the query succeeds. We call such a task *type inference* [8]. In this paper we present a type inference method for Prolog programs. After summarising preliminary materials in section 2, we describe type inference algorithms for Prolog programs in section 3. An application to verification is shown in section 4. See [7] for more details.

## 2. Preliminaries

In the fallowings, we assume familiarity with the basic terminologies of first order logic such as term.atom,substitution and most general unifier(mgu). We also assume knowledge about semantics of Prolog such as Herbrand universe H, Her brand interpretation /,minimum Herbrand Model Mo and transformation *T* of Herbrand interpretation associated with Prolog programs (see [11,14]). We follow the syntax of DEC-10 Prolog [10]. An atom $p\{X_1,X_2,...,X_n)$ is said to be *in general form* when Xi,Xa,..., $X_n$ are distinct variables. A substitution *a* is called *substitution* away *from* an atom *A* when *a* instanciates each variable *X* in *A* to a term *t* such that every variable in *t* is a fresh variable not in *A*.

### 2.1. Definition of Data Types

We introduce type into Prolog to separate definite clauses for data structures from others for procedures,e.g.,

```
type.
   list([ ]).
   list([X|L]) :- list(L).
end.
```

type defines a unary relation by definite clauses. The head of the definite clause takes a term defining a data structure as the argument, either a constant $b$ called *bottom element* or a term of the form $c(t_1, t_2, \ldots, t_n)$ where $c$ is called *constructor*. The body shows a type condition about the proper subterms of the argument.

Here note the set of ground terms prescribed by type predicates. The set of all ground terms $t$ such that $?\text{-}p(t)$ succeeds is called *type of* $p$ and denoted by $\underline{p}$.

*Example 2.1.* Let the definition of a data type *number* be

```
type.
   number(0).
   number(s(X)) :- number(X).
end.
```

Then $\underline{number}$ is a set $\{0, s(0), s(s(0)), \ldots\}$.

Suppose there are defined $k$ data types $p_1, p_2, \ldots, p_k$ and $\underline{p_1}, \underline{p_2}, \ldots, \underline{p_k}$ are disjoint. We denote the set of all ground terms contained in no $\underline{p_i}$ by $\underline{others}$ and consider it like one of types. Then the Herbrand universe $H$ is divided into $k+1$ disjoint sets as follows.

$$H = \underline{p_1} \uplus \underline{p_2} \uplus \ldots \uplus \underline{p_k} \uplus \underline{others}.$$

Procedures are defined following the syntax of DEC-10 Prolog [10],e.g.,

```
append([ ],K,K).
append([X|L],M,[X|N]) :- append(L,M,N).
```

Throughout this paper, we use $P$ as a finite set of definite clauses defining data types and procedures. We assume variables in each definite clause are renamed at each use so that there occurs no variable names conflict.

### 2.2. A Fundamental Theorem for Type Inference in Prolog

Let $I$ and $J$ be Herbrand interpretations. $I$ is said to *cover success set under a restriction* $J$ when it contains the intersection of the minimum Herbrand model $M_0$ and $J$.

An Herbrand interpretation $J$ is said to be *closed with respect to* $P$ when for any ground instance of definite clause in $P$ such that the head is in $J$, any ground atom in the body is also in $J$. This means $M_0 \bigcap J$ is computable within it.

**Theorem** If $J$ is closed with respect to $P$, $T(I) \subseteq T'(I)$ and $T'(I) \bigcap J \subseteq I$, then $I$ covers success set under $J$.

*Proof.* $T(I) \bigcap J \subseteq I$ from $T(I) \subseteq T'(I)$. Let $T_J$ be a monotone transformation of Herbrand interpretations such that $T_J(I)$ is $T(I) \bigcap J$ for any $I$. $T_J$ has a least fixpoint $\bigcap_{T_J(I) \subseteq I} I$ by the Knaster-Tarski fixpoint theorem ([1] p.843,

Theorem 2.1). Because $J$ is closed, $M_0 \bigcap J$ is a fixpoint of $T_J$. Moreover it is the least fixpoint, since $M_0 \bigcap J = \bigcup_{i=0}^{\infty} T_J^i(\emptyset)$ ([1] p.843,Theorem 2.2). Therefor $M_0 \bigcap J \subseteq I$ for any $I$ satisfying $T_J(I) \subseteq I$.

Our goal of type inference is to describe an Herbrand interpretation $I$ covering success set under a restriction $J$ in terms of the types. This is performed by defining an appropriate transformation $T'$ satisfying the theorem above.

## 3. Type Inference in Prolog

In this section, we show how to describe a class of Herbrand interpretations in terms of the types first. Then we define an appropriate transformation satisfying the condition in the theorem in 2.2. The basic computation method and the incremental version are presented in 3.4 and 3.5.

### 3.1. Interpretation by Type

(1) Type Set

A set of ground terms represented by a union of types is called *type set*. Type sets are denoted by $\underline{t}, \underline{t_1}, \underline{t_2}, \ldots$ etc.

*Example 3.1.1.* $\underline{number} \bigcup \underline{list}$ is a type set. $\emptyset$ is a type set,too. $\underline{p_1} \bigcup \underline{p_2} \bigcup \ldots \bigcup \underline{p_k} \bigcup \underline{others}$ is a type set.We denote it by $\underline{any}$. ($\underline{any}$ is not a type but an abbreviation of a type set.)

(2) Type Substitution

An assignment of type sets to variables

$$\Sigma = < X_1 \Leftarrow \underline{t_1}, X_2 \Leftarrow \underline{t_2}, \ldots, X_n \Leftarrow \underline{t_n} >$$

is called *type substitution*. A type assigned to a variable $X$ by $\Sigma$ is denoted by $\Sigma(X)$. We assume $\Sigma(X) = \underline{any}$ for any variable $X$ not appearng explicitly in the domain of $\Sigma$. A type substitution $\Sigma = < X_1 \Leftarrow \underline{t_1}, X_2 \Leftarrow \underline{t_2}, \ldots, X_n \Leftarrow \underline{t_n} >$ is considered the same as a set of substitutions

$$\{ < X_1 \Leftarrow t_1, \ldots, X_n \Leftarrow t_n > \mid t_1 \in \underline{t_1}, \ldots, t_n \in \underline{t_n} \}.$$

*Example 3.1.2.* $< L \Leftarrow \underline{list} >$ is a type substitution. This is considered the same as a set of substitution $\{ < L \Leftarrow t > \mid t$ is any ground term in $\underline{list}$ $\}$. The empty substitution $<>$ is a type substitution assigning $\underline{any}$ to any variable.

The union of two type substituions $\Sigma_1$ and $\Sigma_2$ is a type substitution $\Sigma$ such that $\Sigma(X) = \Sigma_1(X) \bigcup \Sigma_2(X)$ for any $X$ and denoted by $\Sigma_1 \bigcup \Sigma_2$. The intersection of two type substituions $\Sigma_1$ and $\Sigma_2$ is a type substitution $\Sigma$ such that $\Sigma(X) = \Sigma_1(X) \bigcap \Sigma_2(X)$ for any $X$ and denoted by $\Sigma_1 \bigcap \Sigma_2$.

(3) Interpretation by Type

Let $B_0$ be a head of a definite clause "$B_0 :- B_1, B_2, \ldots, B_m$" in $P$ and $\Sigma$ be a type substitution. Then $\Sigma(B_0)$ is considered a set of ground atoms $\{\sigma(B_0) \mid \sigma \in \Sigma\}$. An Herbrand interpretation $I$ represented by a union of all such forms,i.e.

$$\bigcup_{\text{"}B_0 :- B_1, B_2, \ldots, B_m\text{"} \in P} \Sigma(B_0)$$

is called *interpretation by type*.

*Example 3.1.3.* Let $I$ be an Herbrand interpretation

$<>$ $\{append([\,], K, K)) \bigcup$
$< L \Leftarrow \emptyset, M \Leftarrow \emptyset, N \Leftarrow \emptyset >$ $\{append([X|L], M, [X|N]))$.

Then $I$ is an interpretation by type.This is an Herbrand interpretation { append([ ],$t$,$t$) | $t$ is any ground term }.

### 3.2. Restriction by Type

An Herbrand interpretation $J$ of the form $\bigcup_i \Sigma_i(A_i)$ is called *restriction by type*, where each $A_i$ is not necessarily a head of definite clauses in $P$.

*Example 3.2.* $J = <>$ $(append(N, [A], M)) \bigcup <>$ $(reverse(L, M))$ is a restriction by type.

### 3.3. A Transformation for Type Inference

(1) Computation of Type Set of Superterm and Subterm

When each variable $X$ in a term $t$ is instanciated to a ground term in $\Sigma(X)$, we compute a type set containing all ground instances of $t$ as follows and denote it by $t/\Sigma$.

$$t/\Sigma = \begin{cases} \emptyset, & \Sigma(X) = \emptyset \text{ for some } X \text{ in } t; \\ \Sigma(X), & \text{when } t \text{ is a variable } X; \\ \underline{p}, & \text{when } t \text{ is a bottom element } b \text{ of } p \text{ or} \\ & \text{when } t \text{ is of the form } c(t_1, t_2, \ldots, t_n), \\ & c \text{ is a constructor of } p \text{ and} \\ & t_1/\Sigma, t_2/\Sigma, \ldots, t_n/\Sigma \text{ satisfy} \\ & \text{the type conditions;} \\ \underline{any}, & \text{otherwise.} \end{cases}$$

*Example 3.3.1.* Let $t$ be $[X|L]$ and $\Sigma$ be $< L \Leftarrow \underline{list} >$. Then $t/\Sigma$ is $\underline{list}$. Let $t$ be $[X|L]$ and $\Sigma$ be $<>$. Then $t/\Sigma$ is $\underline{any}$.

When a term $t$ containing an occurrence of a variable $X$ is instanciated to a ground term in $\underline{t}$, we compute a type set containing all ground instances of the occurrence of $X$ as follows and denote it by $X/ < t \Leftarrow \underline{t} >$.

$$X/ < t \Leftarrow \underline{t} > = \begin{cases} \underline{t}, & \text{when } t \text{ is a variable } X; \\ X/ < t_i \Leftarrow \underline{t_i} >, & \text{when } t \text{ is of the form} \\ & c(t_1, t_2, \ldots, t_n), \underline{t} \text{ is } \underline{p}, \\ & c \text{ is a constructor of } p, \\ & X \text{ is in } t_i \text{ and} \\ & \underline{t_i} \text{ is a type set assigned} \\ & \text{to the } i\text{-th argument } t_i; \\ \emptyset, & \text{otherwise.} \end{cases}$$

*Example 3.3.2.* Let $t$ be $[X|L]$ and $\underline{t}$ be $\underline{list}$. Then
$X/ < [X|L] \Leftarrow \underline{list} > = \underline{any}$,
$L/ < [X|L] \Leftarrow \underline{list} > = \underline{list}$.
Let $t$ be $[X|L]$ and $\underline{t}$ be $\underline{number}$. Then
$X/ < [X|L] \Leftarrow \underline{number} > = \emptyset$,
$L/ < [X|L] \Leftarrow \underline{number} > = \emptyset$.

(2) Computation of Covering Type Substitution

Let $B_1, B_2, \ldots, B_m$ be a sequence of atoms and $I$ be an interpretation by type. A type substitution is called *covering type substitution* with respect to $B_1, B_2, \ldots, B_m$ and $I$ when it contains every substitution $\sigma$ such that all ground instances $\sigma(B_1), \sigma(B_2), \ldots, \sigma(B_m)$ are in $I$. A transformation $T'$ is defined using the covering type substitutions.

Let $I$ be an interpretation by type $\bigcup_i \Sigma_i(A_i)$ and $B_1, B_2, \ldots, B_m$ be a sequence of atoms. When $B_1, B_2, \ldots, B_m$ are unifiable with $A_{i_1}, A_{i_2}, \ldots, A_{i_m}$ by an mgu $\tau$, we define a type substitution $\Sigma$ on variables in $B_1, B_2, \ldots, B_m$ as follows. Note that we can assume without loss of generality that $t$ contains no variable in $B_1, B_2, \ldots, B_m$ when a variable $X$ in $B_1, B_2, \ldots, B_m$ is instanciated to $t$ by $\tau$, because the unifiability shows there is no cycle.

When $m = 0$ then $\Sigma = <>$.

When $m > 0$ then

(a) Let $t$ be a term containing variables in $A_{i_j}$ and $X$ be a variable in $B_j$. If $\tau$ substitutes $t$ for $X$, then we assigns $t/\Sigma_{i_j}$ to $X$.

(b) Let $t$ be a term containing an occurrence of $X$ in $B_j$ and $Y$ be a variable in $A_{i_j}$. If $\tau$ substitutes $t$ for $Y$, then we assigns $X/ < t \Leftarrow \Sigma_{i_j}(Y) >$ to the occurrence of $X$.

Then $\Sigma$ assigns the intersection $\bigcap_i t_i$ to $X$ when $t_1, t_2, \ldots$ are computed as type sets at different occurrences of $X$ in $B_1, B_2, \ldots, B_m$. (When $\Sigma$ assigns $\emptyset$ to some variable in $B_1, B_2, \ldots, B_m$, we assume $\Sigma$ assigns $\emptyset$ to any variables.)

By $\lceil \frac{B_{i_1} B_{i_2} \cdots B_{i_m}}{\tau} \rceil$, we denote the union of $\Sigma$ for every possible combination of $A_{i_1}, A_{i_2}, \ldots, A_{i_m}$ and its mgu $\tau$.

*Example 3.3.3.* Let $I$ be a type interpretation
$< > (append([\,], K, K)) \bigcup$
$< L \Leftarrow \emptyset, M \Leftarrow \emptyset, N \Leftarrow \emptyset > (append([X|L], M, [X|N]))$
and $B_1$ be a seqence of atoms (though it is only one atom)
$append(A, B, C)$.
Then There are two possibitity of unification. One is $\tau_1 = < A \Leftarrow [\,], B \Leftarrow K, C \Leftarrow K >$ and the corresponding type substitution is $< A \Leftarrow \underline{list} >$. Another is $\tau_2 = < A \Leftarrow [X|L], B \Leftarrow M, C \Leftarrow [X|N] >$ and $< A \Leftarrow \emptyset, B \Leftarrow \emptyset, C \Leftarrow \emptyset >$ is the corresponding type substitution. Hence by taking their union,we have
$\lceil \frac{append(A,B,C)}{I} \rceil = < A \Leftarrow \underline{list} >$

(3) A Transformation $T'$

We define $T'$ as folows.

$$T'(I) = \bigcup_{"B_0 :- B_1, B_2, \ldots, B_m" \in P} \lceil \frac{B_1 B_2 \cdots B_m}{I} \rceil (B_0)$$

It is obvious that $T(I) \subseteq T'(I)$ and $T'$ is monotone for interpretations by type.

### 3.4. Computation of Type Inference

An interpretation by type covering success set under a restriction $J$ is called *type inference under* $J$. The theorem in 2.2 holds for interpretations by type as well. We already have an appropriate transformation $T'$ so that we can compute a type inference under a closed restriction $J$. The outlook of the basic algorithm for type inference is similar to the bottom-up computation of minimum Herbrand models.

$i := -1$; $I_0 := \emptyset$;
**repeat** $i := i + 1$; $I_{i+1} := T'(I_i) \bigcap J$ **until** $I_{i+1} \subseteq I_i$
**return** $I_i$;

Figure 1. Computation of Type Inference

In order to compute type inference under a closed restriction $J$, we need $T'(I) \bigcap J$,which is obtained by using $\lceil \frac{B_1 B_2 \cdots B_m}{I} \rceil (B_0) \bigcap \lceil \frac{B_2}{I} \rceil (B_0)$. $\lceil \frac{B_2}{I} \rceil (B_0)$ is common to all repetitions of the type inference process and can be computed once and for all before the repetitions.

### 3.5. Incremental Type Inference

An atom $A$ is said to be *closed with respect to* $P$ when for any ground instance of the definite clause in $P$ such that the head is a ground instance of $A$, any recursive call in the body is also a ground instance of $A$. (Hence any non-recursive definite clause and any definite clause with a head nonunifiable with $A$ are negligible.) This means the set of ground atoms in $M_0$ of the form of instance of $A$ is computable by some instances of definite clauses. Note that $p(X_1, X_2, \ldots, X_n)$ in general form is always closed.

*Example 3.5.1.* An atom $append(N, [X], M)$ is closesd. This means $\{append(t_1, [t_2], t_3) | t_1, t_2$ and $t_3$ are ground terms$\} \bigcap M_0$ is computable by

```
append([ ],[Y],[Y]).
append([X|L],[Y],[X|N]) :- append(L,[Y],N).
```

The closedness of an atom $A$ can be checked as follows.

(a) Check whether the head $B_0$ is unifiable with $A$ by a substitution for $A$ away from $A$ (see section 2). If it is, decompose the mgu to $\sigma \circ \tau_0$ where $\sigma$ is the restriction to variables in $B_0$ and $\tau_0$ is the restriction to variables in $A$. If it is not,neglect the definite clause.

(b) Check whether each instance of the recursive call in the body $\sigma(B_i)$ is an instance of $A$ and if it is,compute the instanciation $\tau_i$. If it is not,$A$ is not closed.

The set of all instances of definite clauses by $\sigma$ is called *instanciated program* for $A$.

*Example 3.5.2.* Let the atom $A$ be $append(A, [U], C)$. Then the first head $append([\,], L, L)$ is unifiable with $append(A, [U], C)$ by $< L \Leftarrow [Y] > \circ < A \Leftarrow [\,], U \Leftarrow Y, C \Leftarrow [Y] >$. The second head $append([X|L], M, [X|N])$ is unifiable with $append(A, [U], C)$ by $< M \Leftarrow [Y] > \circ < A \Leftarrow [X|L], U \Leftarrow Y, C \Leftarrow [X|N]$ and the instance $append(L, [Y], N)$ in the body is also an instance of $append(A, [U], C)$ by $< A \Leftarrow L, U \Leftarrow Y, C \Leftarrow N >$.

```
append([ ],[Y],[Y]).
append([X|L],[Y],[X|N]) :- append(L,[Y],N).
```

is the instancated program for $append(A, [U], C)$.

An atom satisfying the following condition is called *closure* of $A$ with respect to $P$ and denoted by $\overline{A}$.

(a) $\overline{A}$ is closed with respect to $P$,
(b) $A$ is an instance of $\overline{A}$ and
(c) $\overline{A}$ is an instance of any $\overline{A}'$ satisfying (a) and (b).

*Example 3.5.3.* $reverse(A, B')$ is a closure of $reverse(A, [V|B])$.

The closure is unique up to renaming and $A$ is closed iff $A = \overline{A}$ modulo renaming. (See [7] for the proof of uniqueness and the algorithm to compute the closure.)

Now suppose we would like to compute type inference about $p$ under a restriction by type $<> \overline{p(t_1, t_2, \ldots, t_n)}$ and denote it by $T(\overline{p(t_1, t_2, \ldots, t_n)})$. Let $A_1, A_2, \ldots, A_l$ be non-recursive calls in the bodies of instanciated program for $\overline{p(t_1, t_2, \ldots, t_n)}$. (If some $B_i = q(s_1, s_2, \ldots, s_m)$ and $B_j = q(s'_1, s'_2, \ldots, s'_m)$, we distinguish the predicate symbols by $q_1$ and $q_2$ and assume both of them have the same definite clause program as $q$.) Then we compute $T(\overline{p(t_1, t_2, \ldots, t_n)})$ by initialising $I_0$ to $T(\overline{A_1}) \uplus T(\overline{A_2}) \uplus \cdots \uplus T(\overline{A_l})$, where each $T(\overline{A_i})$ is computed recursively.

$i := -1$; $I_0 := T(\overline{A_1}) \uplus T(\overline{A_2}) \uplus \cdots \uplus T(\overline{A_l})$;
**repeat** $i := i + 1$; $I_{i+1} := T'(I_i)$ **until** $I_{i+1} \subseteq I_i$
**return** $I_i - I_0$;

Figure 2. Incremental Type Inference

If there is no mutual recursion,this process stops. The base case is the type inference for predicates which call no other predicate,e.g. *append*.

*Example 3.5.4.* Suppose we compute $T(reverse(L, M))$. The computation proceeds by initializing $I_0$ to $T(append(N, [X], M))$ as follows.

$I_0 = <> (append([\,], [Y], [Y]) \cup$
$< L \Leftarrow \underline{list}, N \Leftarrow \underline{list} > (append([X|L], [Y], [X|N]))$
$I_1 = I_0 \cup <> (reverse([\,], [\,]) \cup$
$< X \Leftarrow \emptyset, L \Leftarrow \emptyset, M \Leftarrow \emptyset > (reverse([X|L], M))$,
$I_2 = I_0 \cup <> (reverse([\,], [\,]) \cup$
$< L \Leftarrow \underline{list}, M \Leftarrow \underline{list} > (reverse([X|L], M))$

and $I_3 = I_2$. Hence the type inference of *reverse* is

$<> (reverse([\,], [\,]) \cup$
$< L \Leftarrow \underline{list}, M \Leftarrow \underline{list} > (reverse([X|L), M))$

Recursive computation of $T(\overline{A_i})$ is sometimes unnecessary. For example,when $A_i$ is an atom $q(X_1, X_2, \ldots, X_m)$ in general form and $T(q(X_1, X_2, \ldots, X_m))$ is already computed before, then recomputing it all the way slows down the whole computation. As another example, when some $A_i = q(s_1, s_2, \ldots, s_m)$ and $A_j = q(s'_1, s'_2, \ldots, s'_m)$, we distinguish their predicate symbols by $q_1$ and $q_2$. But if $\overline{q(s_1, s_2, \ldots, s_m)} = \overline{q(s'_1, s'_2, \ldots, s'_n)}$ modulo renaming, it turns out to compute the same result twice and the distinction is useless. In order to avoid the defficiency and acceralate the convergence, we store the results computed before for each closed atom, or more precisely for each instanciated program, and utilize them immediately if possible.

## 4. Verification of Prolog Programs Using Type Information

Type inference is used effectively in our verification system [5],[6],[7]. In verification we sometimes simplify the theorem to be proved by assuming that some atom is true or false. In such a case,some information about variables left in the simplified theorem may be lost and we need to retain it to prove the right theorem. This problem was first noticed by Boyer and Moore [2] in their theorem prover(BMTP) for pure LISP. The same problem arises in verification of Prolog programs.

*Example 4.* Suppose we prove a theorem

$\forall U, V, C(\exists B reverse(C, [V|B]) \supset \exists B' reverse([U|C], [V|B']))$.

Because of the definition of *reverse*, it is transformed to

∀ U,A,V,C (∃B reverse(C,[V|B]) ⊃
∃B',D (reverse(C,D) ∧append(D,[U],[V|B'])))

Now let us decide $D$ to be $[V|B]$. This decision is sound and we have a new subgoal

∀ U,V,C,B (reverse(C,[V|B]) ⊃
∃B' (reverse(C,[V|B]) ∧append([V|B],[U],[V|B'])))).

Here we can utilize the antecedant. If $reverse(C, [V|B])$ is false,the theorem is trivially true. Hence we can only need to consider the case $reverse(C, [V|B])$ is true. By replacing $reverse(C, [V|B])$ with true,we have a new subgoal

∀ U,V,B ∃B' append([V|B],[U],[V|B'])

which is further transformed to

∀ U,B ∃B' append(B,[U],B')

because $append([V|B], [U], [V|B'])$ iff $append(B, [U], B')$. But this transformation has generated a too strong theorem and it is in fact not true. (For example, an instance $\forall U \exists B' append(0, [U], B')$ is wrong.) In order to keep the theorem right, we need to add type information as antecedants,i.e. when we derive a subgoal assuming $reverse(C, [V|B])$ true, we have information $list(B)$. Our new subgoal should be

∀ U,V,B (list(B) ⊃ ∃B' append([V|B],[U],[V|B']))

This is provable by induction and we complete the proof.

## 5. Discussions

Several works are done for type inference in Prolog [3],|8),[9] from different point of views. In our approach, both syntactical and semantical concepts appear. It is semantical whether a type inference / covers a success set, while it is syntactical and closely related to the well-typedness in [3],[9] whether a restriction $J$ is closed. Moreover they are related strongly through the crucial condition that $J$ must be closed for / to be computed. Though our approach is still monomorhic, it is new in the following respects.

(a) Our type inference describes a superset of the success set by associating a type substitutions with each definite clause, which gives some accuracy to the types infered.
(b) Our approach solves the problem under a syntactical restriction, which is utilized to infer types more minutely.
(c) Our type inference is not restricted to that for arguments. Type inference can be done for any variables in any procedure call which is not necessarily in general form.

## 6. Conclusions

We have shown a type inference method for Prolog programs and its application to verification. This type inference method is an element of our verification system for Prolog programs developed in 1984.

## Acknowledgements

Our verification system is a subproject of the Fifth Generation Computer System(FGCS) project. The authors would like to thank Dr.K.Fuchi (Director of ICOT) for the chance of this research and Dr.K.Furukawa(Chief of ICOT 2nd Laboratory) and Dr.T.Yokoi(Chief of ICOT 3rd Laboratory) for their advices and encouragements.

## References

[1] Apt,K.R. and M.H.van Emden,"Contribution to the Theory of Logic Programming",J.ACM, Vol.29,No.3,pp 841-862,1982.
[2] Boyer,R.S. and J.S.Moore,"A Computational Logic",Chap. 6., Academic Press, 1979.
[3] Bruynooghe,M.,"Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs", Proc.lst International Logic Programming Conference,pp.129-133,1982.
[4] van Emden,M.H. and R.A.Kowalski,"The Semantics of Predicate Logic as Programing Language",J.ACM,V61.23,No.4, pp.733-742,1976.
[5] Kanamori, T, "Verification of Prolog Programs Using An Extension of Execution",ICOT TR-096,1984.
[6] Kanamori,T.and H.Fujita, "Formulation of Induction Formulas in Verification of Prolog Programs",ICOT TR-094,1984.
[7] Kanamori,T.and K.Horiuchi,"Type Inference in Prolog and Its Applications",ICOT TR-095,1984.
[8] Mishra,P.,"Towards a Theory of Types in Prolog",Proc. 1984 International Symposium on Logic Programming, pp. 289-298,1984.
(9] Mycroft,R.and R.A.O'Keefe,"A Polymorphic Type System for Prolog", Artificial Intelligence 23,pp.295-307,1984.
[10] Pereira,L.M.,F.C.N.Pereira and D.H.D.Warren, "User's Guide to DECsystem-10 Prolog", Occational Paper 15,Dept.of Artificial Intelligence, Edinburgh, 1979.