ICOT Technical Report: TR-095

TR-095

Type Inference in Prolog and Its Applications

Tadashi Kanamori and Kenji Horiuchi
(Mitsubishi Electric Corp.)

December, 1984

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Type Inference in Prolog and Its Applications

Tadashi KANAMORI,Kenji HORIUCHI

Mitsubishi Electric Corporation
Central Research Laboratory
Tsukaguchi-Honmachi 8-1-1
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 provides some accuracy to the types infered. The computation of the superset is done by sequential approximation, which is substantially equivalent to solving inequalities. We also develop an incremental method to utilize already obtained results. We show several interesting applications of type inference to debugging and verification of Prolog programs.

Keywords : Type Inference,Prolog,Program Analysis.

Contents

1. Introduction

It provides usefull information not only to programmers but also to meta-processing systems to infer characteristics of executuion-time behaviors from program texts. Such a task is called program analysis and whether the task is rather easy and effective or not is strongly depends on the target programming language. Logic programming languages are expected to be suitable for such analysis because of their clear semantics (cf.[17]).

In this paper we present a type inference method for Prolog programs, which is one of the important area of program analysis. 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 provides some accuracy to the types infered. The computation of the superset is done by sequential approximation, which is substantially equivalent to solving inequalities. We also develop an incremental method to utilize already obtained results. We show several interesting applications of type inference to debugging and verification of Prolog programs.

After summarizing preliminary materials in section 2, we describe a type inference algorithm for Prolog programs in section 3. Several interesting applications are exemplified in section 4. Lastly in section 5 we discuss the relations to other works ever done.

2. Preliminaries

In the followings, we assume familiarity with the basic terminologies of first order logic such as term,atom(atomic formula),formula,substitution and most general unifier(mgu). We also assume knowledge about semantics of Prolog such as Herbrand universe H,Herbrand interpretation I,minimum Herbrand Model M_0 and transformation T of Herbrand interpretation associated with Prolog programs (see [1],[4],[5],[7],[10]). We follow the syntax of DEC-10 Prolog [15]. As syntactical variables, we use X, Y, Z for variables, s, t for terms and A, B for atoms possibly with primes and subscripts. In addition we use σ, τ, μ, ν for substitutions and Σ for sets of substitutions. An atom $p(X_1, X_2, \ldots, X_n)$ is said to be *in general form* when $X_1, X_2, \ldots, X_n$ are distinct variables. A substitution σ is called a *substitution away from* A when σ instanciates each variable X in A to t such that every variable in t is a fresh variable not in A.

2.1. Definition of Data Types

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

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

type defines a unary relation by definite clauses. The head of definite clause takes a term defining a data structure as the argument, either a constant b called a *bottom element* or a term of the form $c(t_1, t_2, \ldots, t_n)$ where c is called a *constructor*. The body shows a condition of types 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 ?-$p(t)$ succeeds is called a *type* of p and denoted by $\underline{p}$.

Example 2.1: Let the definition of a type *number* be

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

Then $\underline{number}$ is a set $\{0, s(0), s(s(0)), \ldots\}$. ?-$p(t)$ sometimes succeeds without instanciation of variables in t, but we do not include such t in $\underline{p}$. For example,?-$list([X])$ succeeds without instanciation of the variable X. But we do not include $[X]$ in $\underline{list}$ and include only its ground instances.

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.

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

Procedures are defined following the syntax of DEC-10 Prolog [15],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,or conjunction,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, i.e.,$M_0 \bigcap J \subseteq I$.

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 that $M_0 \bigcap J$ 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 types. This is performed by defining an appropriate transformation T' satisfying the theorem above.

2.3. Computation of Minimum Herbrand Model

In order to make the following type inference algorithm understandable, we show a method to generate a sequence of Herbrand interpretations approximating to the minimum Herbrand model M_0.

Computation of Minimum Herbrand Model

$i := -1$; $I_0 := \emptyset$;
repeat $i := i + 1$; $I_{i+1} := T(I_i)$ **until** $I_{i+1} \subseteq I_i$
return I_i;

Here we define the transformation T in a slightly complicated manner.

Let $B_1, B_2, \ldots, B_m$ be a sequence of (conjunction of) atoms and I be an Herbrand interpretation. First we define the set Σ of all substitutions σ such that all ground instances $\sigma(B_1), \sigma(B_2), \ldots, \sigma(B_m)$ are true in I, where we assume that σ substitutes arbitrary ground terms for variables not contained in the domain of σ. We denote the set by $\frac{B_1 B_2 \cdots B_m}{I}$.

Let "B_0 :- $B_1, B_2, \ldots, B_m$" be a definite clause in P. Secondly we define the set of all ground atoms $\sigma(B_0)$ where σ is a substitution defined above for the body of this definite clause,i.e. $\{\sigma(B_0) \mid \sigma \in \frac{B_1 B_2 \cdots B_m}{I}\}$. We denote the set by $\frac{B_1 B_2 \cdots B_m}{I}(B_0)$. Note that the variables not appearing in the body of definite clauses are not included in the domain of the substitutions and can be instanciated to any ground terms.

Example 2.3: Let *append* be defined by the definite clauses mentioned before and I be a set of ground atoms $append(t_1, t_2, t_3)$ where t_1 is a list of the length less than 2 and t_3 is the result of appending t_1 and t_2. Then $\frac{append(L,M,N)}{I}$ is the set of substitution $< L \Leftarrow t_1, M \Leftarrow t_2, N \Leftarrow t_3 >$ where $append(t_1, t_2, t_3) \in I$. And $\frac{append(L,M,N)}{I}(append([X|L], M, [X|N]))$ is the set of ground atoms $append(s_1, s_2, s_3)$ where s_1 is a list of the length less than 3 and s_3 is the result of appending s_1 and s_2.

Then T transforms I to the union of $\frac{B_1 B_2 \cdots B_m}{I}(B_0)$ with respect to all definite clauses in P,i.e.

$$T(I) = \bigcup_{\text{"}B_0 \text{ :- } B_1, B_2, \ldots, B_m\text{"} \in P} \frac{B_1 B_2 \cdots B_m}{I}(B_0)$$

3. Type Inference in Prolog

In this section, we show how to describe a class of Herbrand interpretations in terms of types first. Then we define an appropriate transformation satisfying the condition in the theorem in 2.2. The computation of type inference has a style similar to that of the minimum Herbrand model. Lastly we show how to compute it incrementally utilizing the results already computed before if possible.

3.1. Interpretation by Type

(1) Type Set

A set of ground terms represented by a union of types is called a *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 a *type substitution*. A type assigned to a variable X by Σ is denoted by $\Sigma(X)$. We assume $\Sigma(X) = \underline{any}$ for any variable X not appearng explicitly in the domain of Σ. 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

$\{ < X_1 \Leftarrow t_1, X_2 \Leftarrow t_2, \ldots, X_n \Leftarrow t_n > \mid t_1 \in \underline{t_1}, t_2 \in \underline{t_2}, \ldots, t_n \in \underline{t_n} \}$.

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 Σ_1 and Σ_2 is a type substitution Σ 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 Σ_1 and Σ_2 is a type substitution Σ 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 Σ be a type substitution. Then $\Sigma(B_0)$ is considered as representing 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 an *interpretation by type*.

Example 3.1.3: Let I be an Herbrand interpretation

$< K \Leftarrow \underline{any} >$ $(append([\,], K, K)) \bigcup$

$< X \Leftarrow \underline{any}, L \Leftarrow \emptyset, M \Leftarrow \emptyset, N \Leftarrow \emptyset >$ $(append([X|L], M, [X|N]))$.

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 a *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/Σ.

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

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

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

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 (conjunction of) atoms and I be an interpretation by type. A type substitution is called a *covering type substitution* with respect to $B_1, B_2, \ldots, B_m$ and I when it contains every substitution σ such that all ground instances $\sigma(B_1), \sigma(B_2), \ldots, \sigma(B_m)$ are true 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 τ, we define a type substitution Σ 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 τ, 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 τ substitutes t for X, then we assigns t/Σ_{i_j} to X.

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

Then Σ assigns the intersection $\bigcap_i \underline{t_i}$ to X when $\underline{t_1}, \underline{t_2}, \ldots$ are computed as type sets at different occurrences of X in $B_1, B_2, \ldots, B_m$. When Σ assigns $\emptyset$ to some variable in $B_1, B_2, \ldots, B_m$, we neglect Σ.

By $\lceil \frac{B_1 B_2 \cdots B_m}{I} \rceil$, we denote the union of Σ for every possible combination of $A_{i_1}, A_{i_2}, \ldots, A_{i_m}$ and its mgu τ.

Example 3.3.3: Let I be a type interpretation
$< K \Leftarrow \underline{any} > (append([\,], K, K)) \bigcup$
$< X \Leftarrow \underline{any}, L \Leftarrow \emptyset, M \Leftarrow \emptyset, N \Leftarrow \emptyset > (append([X|L], M, [X|N]))$
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
$< A \Leftarrow \underline{list}, B \Leftarrow \underline{any}, C \Leftarrow \underline{any} >$
Another is $\tau_2 =< A \Leftarrow \overline{[X|L]}, B \Leftarrow \overline{M}, C \Leftarrow [X|N] >$ and
$< A \Leftarrow \emptyset, B \Leftarrow \emptyset, C \Leftarrow \emptyset >$
is the corresponding type substitution. Hence by taking their union,we have
$\lceil \frac{append(A,B,C)}{I} \rceil =< A \Leftarrow \underline{list}, B \Leftarrow \underline{any}, C \Leftarrow \underline{any} >$

(3) A Transformation T'

We define T' as folows. (Note the similarity to the definition of T in 2.3.)

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

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 a *type inference under* J.

Example 3.4.1: Let I be a type interpretation
$< K \Leftarrow \underline{any} > (append([\,], K, K)) \bigcup$
$< X \Leftarrow \underline{any}, L \Leftarrow \underline{list}, M \Leftarrow \underline{any}, N \Leftarrow \underline{any} > (append([X|L], M, [X|N]))$.
Then I covers success set and is a type inference under any restriction.
$<> (append([\,], K, K)) \bigcup <> (append([X|L], M, [X|N]))$
is a type inference as well, but we consider it less accurate.In general
$\bigcup_{"B_0 :- B_1, B_2, \ldots, B_m" \in P} <> (B_0)$
is always a type inference. The interpretation by type in example 3.1.3 is not a type inference under $J = H$.

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 algorithm for basic type inference is similar to that of the computation of the minimum Herbrand model.

Computation of Type Inference

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

In order to compute type inference under a closed restriction J, we need $T'(I) \bigcap J$ in place of $T(I)$ in 2.3. $T'(I) \bigcap J$ is obtained by using $\lceil \frac{B_1 B_2 \cdots B_m}{I} \rceil(B_0) \bigcap \lceil \frac{B_0}{J} \rceil(B_0)$. $\lceil \frac{B_0}{J} \rceil(B_0)$ is common to all repetition of the type inference process and can be computed once and for all before repetition.

Example 3.4.2: Suppose we compute a covering type substitution with respect to a unit clause $append([\,], K, K)$. and $I = \emptyset$ under a restriction by type $J = <> (append(N, [X], M)$. Then because $\lceil \frac{append([\,],K,K)}{J} \rceil$ is $< K \Leftarrow \underline{list} >$ and $\frac{()}{I}$ is $<>$, $< K \Leftarrow \underline{list} >$ is the covering type substitution under J.

First we show an example with no restriction, i.e., $J = \biguplus_p <> (p(X_1, X_2, \ldots, X_n)) = H$. In general,types of predicates calling no other predicate are infered in this manner.

Example 3.4.3: Let *append* be defined by the definition mentioned before and let us compute the basic type inference using the previous algorithm. First we set the initial type interpretation I_0.

$I_0 = < K \Leftarrow \emptyset > (append([\,], K, K)) \bigcup$
$< X \Leftarrow \emptyset, L \Leftarrow \emptyset, M \Leftarrow \emptyset, N \Leftarrow \emptyset > (append([X|L], M, [X|N]))$

Then we compute $T'(I_0)$ as follows.
(a) There is no body for the first definite clause $append([\,], K, K)$. Hence the set of atoms to be true next by T is included in $< K \Leftarrow \underline{any} > (append([\,], K, K))$.
(b) From the body of the second definite clause,we have $\lceil \frac{append(L,M,N)}{I_0} \rceil = < L \Leftarrow \emptyset, M \Leftarrow \emptyset, N \Leftarrow \emptyset >$. Hence the set of atoms to be true next by T is included in $< X \Leftarrow \underline{any}, L \Leftarrow \emptyset, M \Leftarrow \emptyset, N \Leftarrow \emptyset > (append([X|L], M, [X|N])) = \emptyset$.
We have

$I_1 = < K \Leftarrow \underline{any} > (append([\,], K, K)) \bigcup$
$< X \Leftarrow \underline{any}, L \Leftarrow \emptyset, M \Leftarrow \emptyset, N \Leftarrow \emptyset > (append([X|L], M, [X|N]))$.

A similar comutation proceeds and

$I_2 = < K \Leftarrow \underline{any} > (append([\,], K, K)) \bigcup$
$< X \Leftarrow \underline{any}, L \Leftarrow \underline{list}, M \Leftarrow \underline{any}, N \Leftarrow \underline{any} > (append([X|L], M, [X|N]))$

I_3 is computed as well, but now $I_3 = I_2$ and the algorithm stops. The basic type inference obtained is

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

Next we show an example under some restriction.

Example 3.4.4: Suppose we infer the types of *reverse* under a closed restriction by type $J = <> (append(N, [X], M) \bigcup <> (reverse(L, M))$ where

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

The process proceeds as follows.

$I_0 = \emptyset$,
$I_1 = < K \Leftarrow \underline{list} > (append([\,], K, K) \bigcup$
$< X \Leftarrow \emptyset, L \Leftarrow \emptyset, M \Leftarrow \emptyset, N \Leftarrow \emptyset > (append([X|L], M, [X|N])) \bigcup$
$<> (reverse([\,], [\,]) \bigcup$
$< X \Leftarrow \emptyset, L \Leftarrow \emptyset, M \Leftarrow \emptyset > (reverse([X|L], M))$,
$I_2 = < K \Leftarrow \underline{list} > (append([\,], K, K) \bigcup$
$< X \Leftarrow \underline{any}, L \Leftarrow \underline{list}, M \Leftarrow \underline{list}, N \Leftarrow \underline{list} > (append([X|L], M, [X|N])) \bigcup$
$<> (reverse([\,], [\,]) \bigcup$
$< X \Leftarrow \underline{any}, L \Leftarrow \underline{list}, M \Leftarrow \underline{list} > (reverse([X|L], M))$
and $I_3 = I_2$. Note that in the computation of I_1, $< K \Leftarrow \underline{any} > (append([\,], K, K))$ is replaced with $< K \Leftarrow \underline{list} > (append([], K, K))$ and in the computation of I_2, $< X \Leftarrow \underline{any}, L \Leftarrow \underline{list}, M \Leftarrow \underline{any}, N \Leftarrow \underline{any} > (append([X|L], M, [X|N]))$ is replaced with $< X \Leftarrow \underline{any}, L \Leftarrow \underline{list}, M \Leftarrow \underline{list}, N \Leftarrow \underline{list} > (append([X|L], M, [X|N]))$. Hence the type inference under $\overline{J}$ is I_2.

Remark: One might think that solving the inequality $T'(I) \subseteq I$ directly is more efficient than the sequential approximation stated in example 3.4.3. But carefull check of the process of solving inequalities shows that a substantially same computation is performed there.

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 is always inducible. Any definite clause with a head nonunifiable with A is also inducible.) 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: Let the atom A be $append(N, [X], M)$. Then A is closesd. This means that $\{append(t_1, [t_2], t_3) \mid t_1, t_2$ and t_3 are ground terms$\} \bigcap M_0$ is computable by some instances of definite clauses,i.e.,

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

The closedness 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 σ is the restriction to variables in B_0 and τ_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 τ_i. If it is not,A is not closed.

The set of all instances of definite clauses by σ 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 said to be a *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 appendix for the proof of uniqueness and 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 $A_1, A_2, \ldots, A_i$ 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 initializing I_0 to $T(\overline{A_1}) \uplus T(\overline{A_2}) \uplus \cdots \uplus T(\overline{A_i})$, where each $T(\overline{A_i})$ is computed recursively.

Incremental Type Inference

$T(\overline{p(t_1, t_2, \ldots, t_n)})$
  $i := -1$; $I_0 := T(\overline{A_1}) \uplus T(\overline{A_2}) \uplus \cdots \uplus T(\overline{A_i})$;
  repeat $i := i + 1$; $I_{i+1} := T'(I_i)$ **until** $I_{i+1} \subseteq I_i$
  return $I_i - I_0$;

If there is no mutual recursion,this process stops. The base case is the type inference under H in example 3.4.3, where the predicate calls no other predicate.

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

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

and $I_3 = I_2$. Hence the type inference of $reverse$ is $I_2 - I_0$,i.e.,

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

Recursive computation of $T(\overline{A_i})$ is sometimes unnecessary and useless. 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 $\overline{q(s_1, s_2, \ldots, s_n)} = \overline{q(s'_1, s'_2, \ldots, s'_n)}$ 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 of the approximation, we store the results computed before for each closed atom, or more precisely for each instanciated program, and utilize them immediately if possible.

4. Applications of Type Inference

4.1. Debugging of Prolog Programs by Type Inference

We can utilize type inference for debugging by detecting difference between the model intended in programmer's mind and the types infered. Type inference informs us of conditions for variables in a query in terms of data types when the execution of the query succeeds. Suppose we have an interactive system pretending the execution as follows.

The system accepts a query like ??-A where each A is an atom and "??-" is used to distinguish it from the ususal execution mode "?-" in DEC-10 Prolog. Then the system computes the type inference $I = T(A)$ and returns the answer [I].

The system sometimes informs us of existence of a miscoding.

Example 4.1.1: Suppose we have defined *revbad* by

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

intending the usual reverse relation [12] and give a query

```
??-revbad(L,M).
```

Then the systm reply

L=*list*,
M=*any*.

which is different from the model we have in our mind for the correct reverse relation. It also informs us of another kind of miscoding when type inference I consists of a part of the form $< X_1 \Leftarrow \emptyset, X_2 \Leftarrow \emptyset, \ldots, X_n \Leftarrow \emptyset > (B_0)$, because it is likely erroneous that some definite clause never succeeds.

It gives some accuracy to the types infered to describe a superset of the success set by associating type substitutions with each head of definite clause.

Example 4.1.2 : Let <,> and *empty* be predicates defined by

```
0<s(Y).
s(X)<s(Y) :- X<Y.
s(X)>0.
s(X)>s(Y) :- X>Y.
empty(X,Y) :- X<Y,X>Y.
```

Then a query

```
??-empty(X,Y).
```

returns an answer

X=$\emptyset$,
Y=$\emptyset$.

because the computation of I proceeds as follows.

$I_0 = < Y \Leftarrow \overline{any} > (0 < s(Y)) \bigcup < X \Leftarrow \emptyset, Y \Leftarrow \emptyset > (s(X) < s(Y)) \bigcup$
$< X \Leftarrow \overline{any} > (s(X) > 0) \bigcup < X \Leftarrow \emptyset, Y \Leftarrow \emptyset > (s(X) > s(Y)) \bigcup$
$< X \Leftarrow \overline{\emptyset}, Y \Leftarrow \emptyset > (empty(X,Y))$

and $I_1 = I_0$. This example was given to illustrate the imprecision inherent to Mishra's type

inference system [12] by himself, though his system does not need explicit type declarations. The conventional type inference algorithm cannot detect the falsity of $empty(X, Y)$ and usually returns $X = \underline{number}$ and $Y = \underline{number}$.

It is usefull that the query can be atoms of various forms.

Example 4.1.3: Suppose we have a program

```
divide(X,D,0,X) :- X<D.
divide(X,D,s(Q),R) :- X≥D,subtract(X,D,Y),divide(Y,D,Q,R).
0<s(Y).
s(X)<s(Y) :- X<Y.
X≥0.
s(X)≥s(Y) :- X≥Y.
subtract(X,0,X).
subtract(s(X),s(Y),Z) :- subtract(X,Y,Z).
```

Then the query of the general form ??-divide(X,D,Q,R). only receives X=*number*, D=*any*, Q=*number*, R=*number* and D is not necessarily *number*. But more specific inspection is possible by

```
??-divide(X,D,s(Q),R).
```

and we know that when the quotient is non-zero,

X=*number*,
D=*number*,
Q=*number*,
R=*number*.

This is not so trivial to see and it takes a time for programmers to confirm it.

4.2. Verification of Prolog Programs Using Type Information

Type inference is used effectively in our verification system under development [8],[9]. In verification we sometimes simplifies the theorem to be proved by assuming that some atom is true. 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 4.2: Suppose we prove a theorem ∀ U,V,C (∃B reverse(C,[V|B]) ⊃∃B′ reverse([U|C],[V|B′]))
The goal is transformed as follows.

∀ U,V,C (∃B reverse(C,[V|B]) ⊃∃B′ reverse([U|C],[V|B′]))
  ⇓ because of the definition of reverse
∀ 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 which is transformed as follows.

∀ U,V,B ∃B′ append([V|B],[U],[V|B′])
  ⇓ because append([V|B],[U],[V|B′]) iff append(B,[U],B′)
∀ U,B ∃B′ 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 $list(B)$ and type information about variables retained in the new subgoal should be kept. Our new subgoal should be

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

This is provable by induction and we complete the proof.

5. Discussions

Most of the investigations of type inference have been for functional programs [2],[11],[13],[16],[18]. But a few works are done for Prolog [3],[6],[12],[14] from different point of views. Introduction of two dimensions makes easy to classify them, i.e.,syntactical-semantical and monomorphic-polymorphic.

Bruynooghe [3] proposed to introduce types to Prolog in order to enhance reliability and readability and Mycroft and O'Keefe [14] extended the Milner's type polymorhism to Prolog. Though they are mainly concerned with the consistency of type assignments (well-typedness), type inference problem is to decide consistent type assignments for new procedure definitions assuming type assignments of composing predicates and function symbols, if we dare to say. Both of the approaches are polymorphic and syntactical, i.e.,it has no relation to whether the execution of atom with the predicate symbol succeeds or not. The syntactical approach is characterized by its slogan "Well-typed programs do not go wrong".

Mishra [12] takes another approach recently, where the type inference problem is to describe a superset of the arguments of succeeding goals by some expressions from new procedure definitions assuming no explicit definition of data types or type declaration of procedures. (Mishra used some regular expressions.) His approach is monomorhic and semantical. The semantical approach is characterized by its slogan "Ill-typed program cannot succeed".

In our approach, both syntactical and semantical concepts appear. It is semantical whether a type inference I covers a success set, while it is syntactical and closely related to the well-typedness in [3],[14] whether a restriction J is closed. Moreover they are related strongly through the crucial condition that a restriction J must be closed for a type inference I 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 provides 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 $p(X_1, X_2, \ldots, X_n)$.

6. Conclusions

We have shown a type inference method for Prolog programs and its interesting applications to debugging and verification. This type inference method is an element of our verification system for Prolog programs under development.

Acknowledgements

Our verification system is a subproject of Fifth Generation Computer System(FGCS) "Intelligent Programming System". 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.

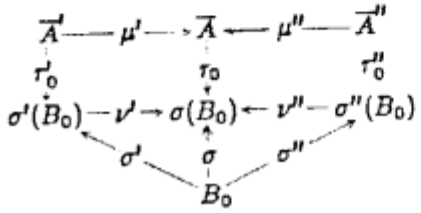
References

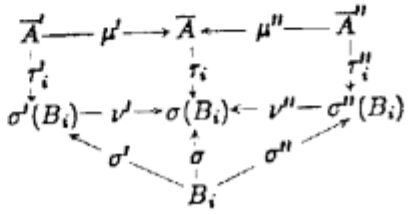
[1] Apt,K.R. and M.H.van Emden,"Contribution to the Theory of Logic Programming",J.ACM, Vol.29,No.3,pp 841-862,1982.
[2] Boyer,R.S. and J.S.Moore,"A Computational Logic",Chap.6.,Academic Press,1979.
[3] Bruynooghe,M.,"Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs", Proc.1st International Logic Programming Conference,pp.129-133,1982.
[4] Clark,K.L.,"Negation as Failure",in Logic and Database (H.Gallaire and J.Minker Eds),pp 293-302,1978.
[5] van Emden,M.H. and R.A.Kowalski,"The Semantics of Predicate Logic as Programing Language",J.ACM,Vol.23,No.4,pp 733-742,1976.
[6] Furukawa,K.,A.Takeuchi,S.Takagi and T.Chikayama,"Type Inference in Logic Programming Language and Its Application to Type Checking",(in Japanese) Proc. of National Conference of Japan Information Processing Society '83,Spring, pp.35-36,1983.
[7] Jaffar,J.,J-L.Lassez and J.Lloyd,"Completeness of the Nagation as Failure Rule", Proc. IJCAI83, Vol.1, pp.500-506, 1983.
[8] Kanamori,T.and H.Seki,"Verification of Prolog Programs Using An Extension of Execution",ICOT Technical Report,TR-093,1984.
[9] Kanamori,T.and H.Fujita,"Formulation of Induction Formulas in Verification of Prolog Programs",ICOT Technical Report,TR-094,1984.
[10] Kowalski,R.A.,"Logic for Problem Solving",North Holland,1980.
[11] Milner,R."A Theory of Type Polymorphism in Programming", J. of Computer and Systems Science 17,pp.348-375,1978.
[12] Mishra,P.,"Towards a Theory of Types in Prolog",Proc. 1984 International Symposium on Logic Programming,pp.289-298,1984.
[13] Morris,J.H.,"Lambda-Calculus Models of Programming Language",Ph.D. Thesis,MAC-TR-57,MIT,1968.
[14] Mycroft,R.and R.A.O'Keefe,"A Polymorphic Type System for Prolog", Artificial Intelligence 23,pp.295-307,1984.
[15] 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.
[16] Reynolds,J.C.,"Automatic Computation of Data Set Definitions", Information Processing 68,pp.456-461,North-Holland,Amsterdam,1969
[17] Sato,T.and H.Tamaki,"Enumeration of Success Patterns in Logic Programs", in ICALP 83 (J.Diaz Ed),Lecture Notes in Computer Science 154,pp.640-652, Springer,1983.
[18] Suzuki,N.,"Infering Types in Smalltalk",Conf.Rec.of 7th ACM Symposium on Principles of Programming Languages,pp.187-199,1980.

Appendix. Closure of Atom

Theorem Closure is unique up to renaming.

Proof: Suppose A has two closures $\overline{A}'$ and $\overline{A}''$. Then from the condition (b), they are unifiable. Let its most general instance be $\overline{A} = \mu'(\overline{A}') = \mu''(\overline{A}'')$. Suppose a head of a recursive definite clause B_0 is unifiable with $\overline{A}$ by an m.g.u. $\tau_0 \circ \sigma$. Hence $(\tau_0 \circ \mu') \circ \sigma$ is a unifier of $\overline{A}'$ and B_0 and $(\tau \circ \mu'') \circ \sigma$ is a unifier of $\overline{A}''$ and B_0. Because $\overline{A}'$ and $\overline{A}''$ are closures of A, B_0 is unifiable with $\overline{A}'$ by an m.g.u. $\tau_0' \circ \sigma'$ and unifiable with $\overline{A}''$ by an m.g.u. $\tau_0'' \circ \sigma''$. This means that for some ν' and ν'', $\sigma = \nu' \circ \sigma' = \nu'' \circ \sigma''$. For all i such that B_i is a recursive call, $\sigma(B_i) = \nu' \circ \sigma'(\overline{A}') = \nu' \circ \tau_i'(\overline{A}') = \nu'' \circ \sigma''(\overline{A}'') = \nu'' \circ \tau_i''(\overline{A}'')$. Hence $\sigma(B_i)$ is a common instance of $\overline{A}'$ and $\overline{A}''$. Because $\mu' \circ \mu''$ is an m.g.u. of $\overline{A}'$ and $\overline{A}''$, there exists a substitution τ_i such that $\sigma(B_i) = \tau_i \circ \mu'(\overline{A}') = \tau_i \circ \mu''(\overline{A}'')$. Then τ_i satisfies the condition of the closedness and $\overline{A}$ is a closure of A. Because of the condition (c), $\overline{A}'$ and $\overline{A}''$ are variants. Hence the closure is unique up to renaming.





Computation of $\overline{p(t_1, t_2, \ldots, t_n)}$

$i := -1$; $A_0 := p(t_1, t_2, \ldots, t_n)$; $P_0 :=$ the set of recursive definite clauses defining p;
repeat
  $i := i + 1$; select a recursive definite clause C in P_i fairly;
  if the head of C is unifiable with A_i
  then $A_{i+1} := closure1(A_i, C)$; $P_{i+1} := P_i - \{C\}$;
  else $A_{i+1} := A_i$; $P_{i+1} := P_i$;
until all heads of definite clauses in P_i are not unifiable with A_i
return A_{i+1}

$closure1(A, "B_0 :- B_1, B_2, \ldots, B_m")$;
$i := -1$; $A_0 := A$;
repeat forever
  $i := i + 1$;
  let $\tau_0 \circ \sigma$ be an m.g.u. of A_i and B_0
  where τ_0 and σ are the restrictions to A_i and B_0;
  let $B_0', B_1', B_2', \ldots, B_l'$ be variants of atoms with p in $\sigma(B_0 :- B_1, B_2, \ldots, B_m)$
  without shared variable by an appropriate renaming;
  $B :=$ most specific common generalization of $B_0', B_1', B_2', \ldots, B_l'$;
  if B is an instance of A_i **then** return A_i **else** $A_{i+1} := B$;

where most specific common generalization is the dual of most general common instanciation, i.e., E is a most specific common generalization of $E_1, E_2, \ldots, E_i$ when
(a) $E_1, E_2, \ldots, E_i$ are instances of E,
(b) E is an instance of any E' satisfying (a).
It is easily obtained by comparing the corresponding subexpressions.