

TR-108

Incorporating Generalization Heuristics
into Verification of Prolog Programs

by

H. Seki and T. Kanamori
(Mitsubishi Electric Corp.)

August, 1985

©1985, ICOT

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

Incorporating Generalization Heuristics into Verification of Prolog Programs

Hirohisa SEKI and Tadashi KANAMORI

Mitsubishi Electric Corporation
Central Research Laboratory
Tsukaguchi-Honmachi 8-1-1
Amagasaki, Hyogo, JAPAN 661

Abstract This paper is concerned with the problem of generalization heuristics, which are incorporated into our verification system for Prolog programs. Two kinds of generalization are discussed, that is, a *mechanical generalization* and an *intelligent generalization*. We show that the mechanical generalization which is used in Boyer-Moore's theorem prover (BMTP) can be performed by the simplification rule of our verification system, as well as in the case of cross-fertilization. To the intelligent generalization heuristic, which is not employed in BMTP, we give a generalization scheme which is naturally incorporated into our inference system of the extended-execution style of the Prolog interpreter, and which proves to be effective also for flawed induction schemes.

1. Introduction

This paper is concerned with a verification system for Prolog programs which is currently under development, as one of the subprojects of the FGCS "Intelligent Programming System" [1].

Logic programming is often advocated as a desirable choice for the verification problem because of its clear semantics (e.g., [2]). In the design of our verification system, we have tried to take advantage of Prolog's characteristics and present first order inference in an extended execution style of Prolog interpreter [3].

Not only first order inference but induction is indispensable as a means of proving interesting properties of Prolog programs such as termination and correctness. In mechanical verification using induction, one of the most difficult problems is to discover an appropriate application of induction scheme. In the case of functional language, Boyer-Moore's theorem prover [4] is famous as one of the most powerful theorem provers, and for its automatic application of induction. The excellent performance of their system is due to the many sophisticated heuristics employed in constructing induction proofs. Into our verification system, various kinds of heuristics controlling the verification process have been integrated, most of which are inspired by Boyer-Moore's system and developed to suit the verification of Prolog programs (e.g., [5]).

In this paper, special attention is paid to "Generalization Heuristics" which are applied when a theorem to be proved is too weak and it is necessary and easier to prove a theorem that is stronger than the original weak one. Two kinds of generalization heuristics are discussed. The first generalization heuristic (we call it *mechanical generalization* in this paper) corresponds to the one used in Boyer-Moore's system, and the second (called *intelligent generalization*) is one that is not employed in their system but is left to the user. Some work has been done with respect to functional language to mechanize the intelligent generalization heuristic (e.g., [6], [7]). Our main purpose lies in an attempt to clarify how these two kinds of generalization heuristics can be incorporated into the verification of Prolog programs.

In the next section, we present our formulation of the verification of Prolog programs and summarize some preliminary materials. In section 3, we present our verification methods using induction. In section 4, we explain two kinds of generalization heuristics; first, we illustrate that the mechanical generalization is naturally incorporated by the simplification rule of our verification system; secondly, we show some examples which cannot be proved by simply applying first order inference and induction, and therefore motivate us to incorporate the intelligent generalization heuristic. In section 5, we discuss in detail how the intelligent generalization is performed in the framework of our verification system. Lastly in section 6, we discuss some implementation issues and related work.

2. Formulation of Verification

In this section, we give our formulation of the verification of Prolog programs. In the following, we follow the syntax of DEC-10 Prolog [8] and assume some familiarity to the terminology of Prolog and first order logic (see e.g., [9], [10]).

In our verification system, we treat only pure Prolog, i.e., we impose the following 3 conditions on Prolog programs :

- (i) no negative clauses and no "not" in programs,
- (ii) no executable primitives,
- (iii) no "cut" symbol (!).

On the other hand, our specification language is a subclass of first order logic formulas. That is, a specification is expressed by a closed formula which can be transformed into the following prenex form (we call it an *S-formula*) :

$$\forall X_1 \dots \forall X_m \exists Y_1 \dots \exists Y_n F \quad (m, n \geq 0), \quad \text{where } F \text{ contains no quantifiers.}$$

Let S be a specification and P be a Prolog program and P^* be the completion of P in the sense of Clark [11]. Then we adopt a formulation that verification of S with respect to P proves that

$$P^* \vdash S.$$

The above means that S is a logical consequence of P^* using first order inference and induction, which are discussed in the next section.

Before going into explanation about the details of our verification method, we need some preliminaries and definitions. First, we use the notion of *polarity* ([9], [10]). The positive and negative subformulas are defined as follows :

- (i) \mathcal{F} is a positive subformula of \mathcal{F} .
- (ii) When $\neg \mathcal{G}$ is a positive (negative) subformula of \mathcal{F} , then \mathcal{G} is a negative (positive) subformula of \mathcal{F} .
- (iii) When $\mathcal{G} \wedge \mathcal{H}$ or $\mathcal{G} \vee \mathcal{H}$ is a positive (negative) subformula of \mathcal{F} , then \mathcal{G} and \mathcal{H} are positive (negative) subformulas of \mathcal{F} .
- (iv) When $\mathcal{G} \supset \mathcal{H}$ is a positive (negative) subformula of \mathcal{F} , then \mathcal{G} is a negative (positive) subformula of \mathcal{F} and \mathcal{H} is a positive (negative) subformula of \mathcal{F} .

Next, variables which appear in a specification are distinguished in the following way. Let F be a closed first order formula. When $\forall X G$ is a positive subformula or $\exists X G$ is a negative subformula of F , then X is called a *free variable* of F . On the other hand, when $\exists Y H$ is a positive subformula or $\forall Y H$ is a negative subformula of F , then Y is called an *undecided variable* of F . In other words, when F is transformed into prenex normal form, free variables are variables quantified universally, and undecided variables are those quantified existentially.

A *goal formula* is a formula which is obtained from an S -formula by replacing each undecided variable Y with $?Y$ and deleting all quantifications. A substitution σ for a goal formula G is called a *deciding substitution* when σ instantiates no free variable in G .

Example : An S -formula :

$$\forall X (\text{list}(X) \supset \forall Y \exists Z \text{append}(X, Y, Z))$$

is represented by a goal formula :

$$\text{list}(X) \supset \text{append}(X, Y, ?Z),$$

where $\text{list}(X)$ is a negative atom and $\text{append}(X, Y, ?Z)$ is a positive atom.

Lastly we introduce some notations. A replacement of an occurrence of a term t in a formula F by s is denoted by $F_t[s]$, and a replacement of all occurrences of a term t in a formula F by s is denoted by $F_t(s)$. A formula F is said to be in a *reduced form* [10] with respect to logical constants *true* and *false*, if F is either (i) *true*, or (ii) *false*, or (iii) if neither *true* nor *false* occur in F . The reduced form of a formula F is denoted by $F \downarrow$.

3. Inference Rules

In this section, we give a brief description of our verification procedure for a specification (for a detailed explanation, see [3]).

3-1. Extended Execution

Since a specification is not restricted in a Horn clause but is expressed in an S -formula, we need some extension of the usual Prolog interpreter. For this purpose, our verification system employs the following four inference rules.

(1) Case Splitting

Let G be a goal formula and H either (i) an outermost positive subformula of the form $H_1 \wedge \dots \wedge H_k$ or (ii) an outermost negative subformula of the form $H_1 \vee \dots \vee H_k$ ($k > 1$). Then, if each undecided variable appearing in H_i appears only in H_i ($1 \leq i \leq k$), we generate new k AND-goals $G_H[H_1], \dots, G_H[H_k]$.

The following two rules are an extension of a Prolog interpreter using polarity.

(2) Definite Clause Inference (DCI)

Let A be a positive atom in a goal formula G and let " $B :- B_1, \dots, B_m$ " be any definite clause in P . When A is unifiable with B by a deciding m.g.u. (most general unifier) σ , we generate new OR-goals for all such definite clauses : $\sigma(G_A[B_1 \wedge \dots \wedge B_m]) \downarrow (\sigma(G_A[true]) \downarrow$ when $m=0$) . All newly introduced variables are treated as fresh undecided variables.

(3) Negation as Failure Inference (NFI)

Let A be a negative atom in a goal formula G and let " $B :- B_1, \dots, B_m$ " be any definite clause in P . When A is not unifiable with a head of any definite clause in P , we generate a goal : $G_A[false] \downarrow$. When A is unifiable with B of a definite clause by an m.g.u. σ , we generate new AND-goals for all such definite clauses : $\sigma(G_A[B_1 \wedge \dots \wedge B_m]) \downarrow (\sigma(G_A[true]) \downarrow$ when $m=0$) . All newly introduced variables are treated as fresh free variables.

(4) Simplification

Let G be a goal formula. When A_1, \dots, A_m are positive atoms and A_{m+1}, \dots, A_n are negative atoms unifiable to A by a deciding m.g.u. σ , we generate new AND-goals : $\sigma(G)_A(true) \downarrow$ and $\sigma(G)_A(false) \downarrow$.

These four inference rules are repeatedly applied to a given goal formula until it is reduced to *true* or *false*. If these rules cannot be applied any more, then we appeal to the following induction.

3-2. Induction

Our verification system utilizes inductive proofs which are based on structural induction schemes. Those induction schemes are also used in [12],[13],[14] for the verification of Prolog programs. For example, the following is an induction scheme for list X :

$$\frac{Q([]) \quad \forall A, X (Q(X) \supset Q([A|X]))}{\forall X : list \ Q(X)}$$

where $Q(X)$ is a theorem to be proved, $Q([])$ is a base case, and $\forall A, X (Q(X) \supset Q([A|X]))$ is an induction step.

3-3. Examples

As an example of a verification which illustrates how the above-mentioned inference rules are applied, let us consider the following theorem.

theorem(*associativity-of-append*).
 $\forall XYZLMN \ append(X, Y, L) \wedge append(Y, Z, M) \wedge append(X, M, N) \supset append(L, Z, N).$
end.

where *append* is defined in a usual way :

$append([], X, X).$
 $append([A|X], Y, [A|Z]) :- append(X, Y, Z).$

and let $Q(X)$ be :

$\forall YZLMN \ append(X, Y, L) \wedge append(Y, Z, M) \wedge append(X, M, N) \supset append(L, Z, N).$

Base Case

The proof of base case $Q([])$ is straightforward.

$append([], Y, L) \wedge append(Y, Z, M) \wedge append([], M, N) \supset append(L, Z, N)$
 \Downarrow NFI for $append([], Y, L)$
 $append(Y, Z, M) \wedge append([], M, N) \supset append(Y, Z, N)$
 \Downarrow NFI for $append([], M, N)$
 $append(Y, Z, M) \supset append(Y, Z, M)$
 \Downarrow simplification
true

Induction Step

Similarly in induction step, inference rules are applied to the induction step goal which is obtained from $\forall A, X \ Q(X) \supset Q([A|X])$ by deleting quantifiers carefully with difference between free variables and undecided variables.

$$\begin{aligned} & [append(X, ?Y_0, ?L_0) \wedge append(?Y_0, ?Z_0, ?M_0) \wedge append(X, ?M_0, ?N_0) \supset append(?L_0, ?Z_0, ?N_0)] \\ & \supset [append([A|X], Y, L) \wedge append(Y, Z, M) \wedge append([A|X], M, N) \supset append(L, Z, N)] \\ & \Downarrow \text{NFI for } append([A|X], Y, L) \\ & [append(X, ?Y_0, ?L_0) \wedge append(?Y_0, ?Z_0, ?M_0) \wedge append(X, ?M_0, ?N_0) \supset append(?L_0, ?Z_0, ?N_0)] \\ & \supset [append(X, Y, L_1) \wedge append(Y, Z, M) \wedge append([A|X], M, N) \supset append([A|L_1], Z, N)] \\ & \Downarrow \text{NFI for } append([A|X], M, N) \\ & [append(X, ?Y_0, ?L_0) \wedge append(?Y_0, ?Z_0, ?M_0) \wedge append(X, ?M_0, ?N_0) \supset append(?L_0, ?Z_0, ?N_0)] \\ & \supset [append(X, Y, L_1) \wedge append(Y, Z, M) \wedge append(X, M, N_1) \supset append([A|L_1], Z, [A|N_1])] \\ & \Downarrow \text{DCI for } append([A|L_1], Z, [A|N_1]) \\ & [append(X, ?Y_0, ?L_0) \wedge append(?Y_0, ?Z_0, ?M_0) \wedge append(X, ?M_0, ?N_0) \supset append(?L_0, ?Z_0, ?N_0)] \\ & \supset [append(X, Y, L_1) \wedge append(Y, Z, M) \wedge append(X, M, N_1) \supset append(L_1, Z, N_1)] \\ & \Downarrow \text{simplification w.r.t } append(X, ?Y_0, ?L_0) \text{ and } append(X, Y, L_1) \\ & \Downarrow \text{simplification w.r.t } append(Y, ?Z_0, ?M_0) \text{ and } append(Y, Z, M) \\ & \Downarrow \text{simplification w.r.t } append(X, M, ?N_0) \text{ and } append(X, M, N_1) \\ & \Downarrow \text{simplification w.r.t } append(L_1, Z, N_1) \text{ and } append(L_1, Z, N_1) \\ & \text{true} \end{aligned}$$

In the above proof, we have omitted one of AND-goals which is generated by the simplification rule and is immediately reduced to be true.

4. Generalization Heuristics

In the preceding section, we illustrated how we perform the verification of Prolog programs by using the extended execution and induction. In most cases, however, the verification procedure will not perform satisfactorily, if it is not guided by some sophisticated heuristics, just as in the case of functional language like Boyer-Moore's theorem prover (BMTP for short)[4].

In this section, special attention is paid to the use of "Generalization Heuristics." We discuss two kinds of such generalization heuristics which have been incorporated into our verification system.

4-1. Mechanical Generalization

The generalization employed in BMTP is a heuristic by which a term in a formula is replaced by a variable under an appropriate condition. For example, when the generalization

heuristic is applied to a formula in functional language ;

$$\text{reverse}(\text{reverse}(L)) = L \supset \text{reverse}(\text{append}(\text{reverse}(L), [X])) = [X|L],$$

then the following formula is obtained :

$$\text{reverse}(N) = L \supset \text{reverse}(\text{append}(N, [X])) = [X|L],$$

where term $\text{reverse}(L)$ in the first formula is replaced by a new variable N , and thus a more general formula is obtained. On the other hand, let G be a goal formula :

$$\frac{(\text{reverse}(L, M) \supset \text{reverse}(M, L))}{\supset (\text{reverse}(L, M) \wedge \text{append}(M, [X], N) \supset \text{reverse}(N, [X|L]))}.$$

When we apply the simplification rule to the above underlined positive and negative $\text{reverse}(L, M)$ in G , then we generate the following AND-goals :

$$\begin{aligned} &(\text{true} \supset \text{reverse}(M, L)) \supset (\text{true} \wedge \text{append}(M, [X], N) \supset \text{reverse}(N, [X|L])) \downarrow, \\ &(\text{false} \supset \text{reverse}(M, L)) \supset (\text{false} \wedge \text{append}(M, [X], N) \supset \text{reverse}(N, [X|L])) \downarrow, \end{aligned}$$

which are immediately reduced to :

$$\text{reverse}(M, L) \supset (\text{append}(M, [X], N) \supset \text{reverse}(N, [X|L]))$$

and true , respectively. This inference exactly corresponds to the above-mentioned generalization heuristic employed in BMTP.

Likewise, the inference of cross-fertilization in BMTP also corresponds to the one performed by simplification [3]. In this way, generalization and cross-fertilization which are treated as different heuristics in BMTP, are performed in a unified way by the simplification rule in our verification system.

Furthermore, the heuristic of eliminating destructors in BMTP can be considered as a kind of generalization heuristic. For example, selectors for data structure like $\text{car}(L)$ and $\text{cdr}(L)$ appearing in the goals of BMTP are eliminated and replaced by variables. In Prolog programs, we usually don't use such selectors explicitly ; we do without them by using unification. Hence, Prolog programming style sometimes makes unnecessary such a generalization heuristic as eliminating destructors.

4-2. Intelligent Generalization

The second generalization heuristic differs from the above one, and BMTP intentionally does not employ it because it requires "creative" insight (chap.XII in [4]). In the verification of Prolog programs, however, there are also some cases where, in proving an induction step goal, we cannot use its induction hypothesis because of *mismatching* with its conclusion. As a result, we cannot prove the induction step goal simply by using first order inference and

induction, so we need some kind of heuristic to resolve the mismatching.

As an example of such a case, consider the proof of the following theorem.

`theorem(reverse*-reverse*).`
 $\forall XY \text{ reverse}^*(X, [], Y) \supset \text{reverse}^*(Y, [], X) \quad \dots (G_0),$
`end.`

where program *reverse** is defined as follows :

`reverse*([], X, X).`
`reverse*([A|X], Y, Z) :- reverse*(X, [A|Y], Z).`

First, we try to prove the above theorem by induction and the following induction scheme is generated.

$$\frac{Q([]) \quad \forall A, X (Q(X) \supset Q([A|X]))}{\forall X : \text{list } Q(X)}$$

where $Q(X)$ is $\forall Y \text{ reverse}^*(X, [], Y) \supset \text{reverse}^*(Y, [], X)$.

The proof of its base case, $Q([])$, is trivial. The proof of the induction step goal, however, is not easily performed because of those underlined mismatched literals shown below.

[induction step] $Q(X) \supset Q([A|X])$
 $(\text{reverse}^*(X, [], ?Y_0) \supset \text{reverse}^*(?Y_0, [], X))$
 $\supset (\text{reverse}^*([A|X], [], Y) \supset \text{reverse}^*(Y, [], [A|X]))$
 $\Downarrow \text{NFI for } \text{reverse}^*([A|X], [], Y)$
 $(\text{reverse}^*(X, [], ?Y_0) \supset \text{reverse}^*(?Y_0, [], X))$
 $\supset (\text{reverse}^*(X, [A], Y) \supset \text{reverse}^*(Y, [], [A|X]))$

Here, since we cannot apply simplification because of mismatching between “ $[]$ ” and “ $[A]$ ” in the above underlined parts, there is no way to use the induction hypothesis.

These “phenomena” are not restricted to those goals containing *reverse**, but often happen to the verification of Prolog programs containing, for example, *palin** [15], *fringe**, etc. (see Appendix). As is obviously known, each program with asterisk has an “accumulator” ([6]) like the second argument of *reverse**, which, though it makes the computation linear order, cause at the same time mismatching between an induction hypothesis and its conclusion.

Hence, in order to solve these kinds of mismatching, it is necessary to incorporate some heuristics and generate a generalized goal. Our verification system generates the following

goal mechanically :

theorem (*generalized reverse*-reverse**).
 $\forall X S M T \text{ reverse}^*(X, S, M) \wedge \text{reverse}^*(S, X, T) \supset \text{reverse}^*(M, [], T) \quad \dots (G_{gen}),$
end.

We call such kind of generalization an "*intelligent generalization*." It is easily known that the above theorem is actually a generalized goal of G_0 and its proof can be rather straightforwardly performed.

5. Intelligent Generalization Heuristic

In this section, we state how the intelligent generalization heuristic is applied to the proof of an induction step goal and give the intelligent generalization scheme which mechanically generates its generalized goal. We then go on to show how its scheme is also effective for flawed induction schemes [4].

5-1. Intelligent Generalization Scheme

At first, for ease of understanding, we illustrate the intelligent generalization heuristic by using the previous example.

$$\frac{\text{reverse}^*(X, [], ?Y_0) \supset \text{reverse}^*(?Y_0, [], X)}{\supset (\text{reverse}^*(X, [A], Y) \supset \text{reverse}^*(Y, [], [A|X]))}$$

The first step of intelligent generalization is to find those *mismatching arguments* which make it impossible to use the induction hypothesis. In the above example, those mismatching arguments are "[]" and "[A]" in $\text{reverse}^*(X, [], ?Y_0)$ and $\text{reverse}^*(X, [A], Y)$, respectively.

Next, we replace the mismatching arguments in the induction conclusion by new variables. In the above, "[A]" is replaced by a new variable, say, T . We call those variables contained in the mismatching arguments "*mismatching variables*," and those arguments in the induction conclusion which contain mismatching variables are called "*arguments relating to mismatching*." In this case, " A " is a mismatching variable and " $[A|X]$ " is an argument relating to mismatching. These arguments relating to mismatching are also replaced by new distinct variables. In this case, we replace " $[A|X]$ " by a new variable, say, U , and the induction conclusion becomes the following formula :

$$\text{reverse}^*(X, T, Y) \supset \text{reverse}^*(Y, [], U)$$

which means that $\forall X Y T U \text{ reverse}^*(X, T, Y) \supset \text{reverse}^*(Y, [], U) \quad \dots (G_1)$.

Clearly, the above goal (G_1) is not a correct specification but an "over-generalized" goal

of the original theorem (*reverse*-reverse**), and an appropriate constraint condition should be imposed on (G_1) in order to obtain a correct generalized goal. Hence, we assume some "constraint relation" between freshly introduced variables and those variables which appear in arguments relating to mismatching. In this case, a constraint relation, say, $\mathcal{R}(T, X, U)$, is imposed, because T and U are newly introduced variables and X is a variable which appears in the argument relating to mismatching, i.e., $[A|X]$. We assume the following goal as a generalized goal of the original one :

$$\text{reverse}^*(X, T, Y) \wedge \mathcal{R}(T, X, U) \supset \text{reverse}^*(Y, [], U) \dots (G_2),$$

where $\mathcal{R}(T, X, U)$ is some relation whose precise form is not determined yet ; we call such a goal that contains an unspecified relation a "temporary goal."

The third step is to infer the constraint relation mentioned above. Our current system infers such a relation from the following conditions ; the first inferring condition is called *generalization condition*, which is a necessary condition for a temporary goal to be actually a generalized goal of the original one. In this case, by comparing G_0 and G_2 , it follows that X equals U when S is $[]$, which imposes the following constraint on $\mathcal{R}(T, X, U)$:

$$\mathcal{R}([], X, X). \dots (CR_1)$$

The second inferring condition is that derived from "pseudo verification," which means that we apply the inference rule mentioned in section 3 to the temporary goal. For example, if we apply induction on G_2 , we get the following verification process for the induction step :
 $\forall X \ P(X) \supset P([A|X])$, where $P(X)$ means that
 $\forall Y T U \ \text{reverse}^*(X, T, Y) \wedge \mathcal{R}(T, X, U) \supset \text{reverse}^*(Y, [], U)$.

$$\begin{aligned} & (\text{reverse}^*(X, ?T_0, ?Y_0) \wedge \mathcal{R}(?T_0, X, ?U_0) \supset \text{reverse}^*(?Y_0, [], ?U_0)) \\ & \supset (\text{reverse}^*([A|X], T, Y) \wedge \mathcal{R}(T, [A|X], U) \supset \text{reverse}^*(Y, [], U)) \\ & \Downarrow \text{NFI for } \text{reverse}^*([A|X], T, Y) \\ & (\text{reverse}^*(X, ?T_0, ?Y_0) \wedge \mathcal{R}(?T_0, X, ?U_0) \supset \text{reverse}^*(?Y_0, [], ?U_0)) \\ & \supset (\text{reverse}^*(X, [A|T], Y) \wedge \mathcal{R}(T, [A|X], U) \supset \text{reverse}^*(Y, [], U)) \\ & \Downarrow \text{simplification w.r.t } \text{reverse}^*(X, ?T_0, ?Y_0) \text{ and } \text{reverse}^*(X, [A|T], Y) \\ & (\mathcal{R}([A|T], X, ?U_0) \supset \text{reverse}^*(Y, [], ?U_0)) \\ & \supset (\mathcal{R}(T, [A|X], U) \supset \text{reverse}^*(Y, [], U)) \\ & \Downarrow \text{simplification w.r.t } \text{reverse}^*(Y, [], ?U_0) \text{ and } \text{reverse}^*(Y, [], U) \\ & \mathcal{R}(T, [A|X], U) \supset \mathcal{R}([A|T], X, U). \dots (CR_2) \end{aligned}$$

Similarly, as for the base case of the induction scheme for $P(X)$, we obtain the following constraint relation :

$$\mathcal{R}(Y, [], U) \supset \text{reverse}^*(Y, [], U). \dots (CR_3)$$

Then, from these constraint relations, (CR_1) , (CR_2) and (CR_3) , we try to identify the unknown relation, $R(T, X, U)$. Our verification system searches for an already defined Prolog program which satisfies those constraint relations. Actually, we rewrite those constraint relations into formulas in Horn clauses ;

$$R([], X, X). \quad \dots (CR_1^*)$$

$$R([A|T], X, U) :- R(T, [A|X], U). \quad \dots (CR_2^*)$$

$$reverse^*(Y, [], U) :- R(Y, [], U). \quad \dots (CR_3^*),$$

and then find a definite clause which can be "unified" with those (CR_i^*) 's whose heads are the unknown constraint relation. If there are no such clauses defined in the system, we consider all the permutations of the arguments of the constraint relation. That is, for example in this case, we consider all the permutations of $\{T, X, U\}$ and check each case for $R(T, U, X)$, $R(U, T, X)$, $R(U, X, T)$, etc. In the above example, from (CR_1^*) and (CR_2^*) , we can find that

$$R(T, X, U) \equiv reverse^*(T, X, U), \quad \dots (G_3)$$

and it is easily confirmed to satisfy (CR_3) . From G_2 and G_3 , we finally obtain the generalized goal G_{gen} mentioned in section 4-2.

To sum up, the intelligent generalization scheme consists of the following 4 steps :

- (1) Find out mismatching arguments in literals to which the induction has been applied.
- (2) Replace those arguments relating to the mismatching by new variables in the induction conclusion (and obtain an "over-generalized goal").
- (3) On the over-generalized goal, impose an appropriate constraint relation by pseudo verification and by the generalization condition.
- (4) From the restrictions derived in (3), infer the constraint relation.

As another example of generalization which we think is not so easy for us to find an appropriate generalized goal, consider the next one :

```
theorem(equivalence-of-fringe*-flatten).
   $\forall X : list, F \text{ fringe}^*(X, [], F) \supset \text{flatten}(X, F)$ 
end.
```

When we follow the above-mentioned intelligent generalization scheme, then the following generalized goal is suggested :

$$fringe^*(X, T, F) \wedge S(X, T, U) \supset flatten(U, F) \quad \dots (F_1)$$

where T and U are newly introduced variables, and constraint relation $S(X, T, U)$ is required to satisfy the following conditions :

$$S(X, [], X). \quad \dots (C_1)$$

$$S(X, [Y|T], U) :- S([X|Y], T, U) \quad \dots (C_2)$$

In this case, since our verification system cannot find already defined clauses which match the above (C_1) and (C_2) , these two constraint relations are returned to us. In fact, however, we know that any predicate satisfying both (C_1) and (C_2) is sufficient to make F_1 true.

As for the inference method in step (4) of the generalization scheme, our current implementation deals only with those constraint relations which can be reduced into Horn clauses as mentioned in the examples.

5-2. Application to Flawed Induction Schemes

The application of the intelligent generalization scheme is not restricted to the above-mentioned proofs, but it is sometimes effective also for a "flawed" induction scheme [5]. For example, consider the following example which is a corollary of the associativity of *append*.

theorem(corollary-of-append-associativity).

$$\forall XDR \text{ append}(X, X, D) \wedge \text{append}(X, D, R) \supset \text{append}(D, X, R)$$

end.

From the definition of *append*, we note that the predicate *append*(X, Y, Z) recursively changes its first and third argument and leaves its second argument fixed. If we use the terminology of BMTP, the first and third arguments are *changing variables* and the second argument is an *unchanging variable*. The above example shows a case where an induction scheme suggested by an atom and another induction scheme suggested by a different atom, are mutually flawed. That is, the induction scheme suggested by *append*(X, D, R) recursively changes X which is an unchanging variable in *append*(D, X, R), while the induction scheme suggested by *append*(D, X, R) recursively changes D which is an unchanging variable in *append*(X, D, R) (and note that the induction scheme suggested by *append*(X, X, D) is flawed by itself).

In this case, our intelligent generalization scheme generates the following generalized goal :

$$\text{append}(X, V, D) \wedge \text{append}(X, U, R) \wedge \text{append}(V, V, U) \supset \text{append}(D, V, R)$$

where $append(V, V, U)$ is a constraint relation obtained and we know that the above goal is easily proved by an "unflawed" induction.

6. Concluding Remarks & Related Work

This paper has shown how generalization heuristics are incorporated into the verification system of Prolog programs. Two kinds of generalization, mechanical generalization and intelligent generalization, are discussed. We have shown that the mechanical generalization used in BMTP, can be performed by simplification in our verification system, as well as in the case of cross-fertilization. To the intelligent generalization heuristic, we have given a generalization scheme which is naturally incorporated into our inference system of the extended-execution style, and which has proved to be effective also for flawed induction schemes.

We apparently owe our heuristics to BMTP and related works by Moore [8] and Aubin [9], although there is a difference in target languages and deduction methods. As for inferring constraint relation mentioned in section 5, we find some relevance to Shapiro's [16] Model Inference System which is a general theory to infer a Prolog predicate satisfying given facts. Our intelligent generalization scheme differs in that it gives a Prolog predicate some constraint relations to be satisfied.

The generalization heuristics presented here have been examined by numerous hand proofs and their first versions have been implemented in DEC-10 Prolog on DEC-2060 as one of the heuristics of our verification system for Prolog programs. It is our intention that our verification system will be implemented on PSI [17] as a basis of "Intelligent Programming Environment" for Prolog programming on which we can perform various kinds of experiments such as program transformation and synthesis.

Acknowledgments

The authors appreciate K. Fuchi(Director of ICOT) and K. Furukawa (the Chief of ICOT 2nd Laboratory) for the chance of doing this research. We would also like to thank A. Fusaoka, H. Fujita and K. Suzuki for their useful discussions and help.

Appendix

The predicates defined below are referred in this paper and their "usually" defined ones.

$reverse^*([], X, X).$

$reverse^*([A|X], Y, Z) :- reverse^*(X, [A|Y], Z).$

```

reverse([ ],[ ]).
reverse([A|X],Y) :- reverse(X,Z),append(X,[A],Y).

fringe*(A,[ ],[A]) :- atom(A).
fringe*(A,[X|S],[A|L]) :- atom(A),fringe*(X,S,L).
fringe*([X|Y],S,L) :- fringe*(X,[Y|S],L).

flatten(A,[A]) :- atom(A).
flatten([X|Y],L) :- flatten(X,Lx),flatten(Y,Ly),append(Lx,Ly,L).

palin*(X,X).
palin*(X,[A|X]).
palin*(Y,[A|X]) :- palin*([A|Y],X).

palin([ ]).
palin([A]).
palin([A|X]) :- append(Y,[A],X),palin(Y).

```

References

- [1] Furukawa, K. and T. Yokoi, "Basic Software System" In Proc. FGCS-84, Tokyo, Japan, November, 1984, pp. 37-57.
- [2] Bowen, K. A. "Programming with Full First-Order Logic" In Machine Intelligence 10 (1982) 421-440.
- [3] Kanamori, T. "Verification of Prolog programs Using an Extension of Execution," ICOT Technical Report, TR-096, 1984.
- [4] Boyer, R. S. and J. S. Moore. Computational Logic, New York: Academic Press, 1979.
- [5] Kanamori, T. et al. "Formulation of Induction Formulas in Verification of Prolog Programs," ICOT Technical Report, TR-094, 1984.
- [6] Moore, J. S., "Introducing iteration into the pure LISP theorem prover." IEEE Trans. Software Eng. 1:3 (1975) 328-338.
- [7] Aubin, R. "Some Generalization Heuristics in Proofs by Induction" In Proc. IRIA Colloq. on Proving and Improving Programs. Arc et Senans, France, July, 1975, pp. 197-208.
- [8] Pereira, L. M., F. C. Pereira and D. H. Warren "User's Guide to DECsystem-10 PROLOG," Technical Report, Univ. of Edinburgh, September 1978.
- [9] Prawitz, D. Natural Deduction, A Proof-Theoretical-Study, Stockholm: Almqvist & Wiksell, 1965.
- [10] Murray, N. V., "Completely Non-Clausal Theorem Proving", Artificial Intelligence, 18:1 (1982) 67-85.
- [11] Clark, K. L. "Negation as Failure" In Logic and Database, Gallaire. H and J. Minker. Eds., (1978) 293-322.

- [12] Clark, K. L. and S-Å. Tärnlund "A First Order Theory of Data and Programs" In IEIP-77. Toronto, Canada, August, 1977, pp. 939-944.
- [13] Clark, K. L. "Predicate Logic as a Computational Formalism," Technical Report 59, Imperial College, December 1979.
- [14] Sterling, L. and A. Bundy "Meta-Level Inference and Program Verification" In 6th Conf. on Automated Deduction. Lecture Notes in Computer Science 138, 1982, pp. 144-150.
- [15] Hogger, C., "Derivation of Logic Programs." JACM 28 : 2 (1981) 372-422.
- [16] Shapiro, E. Y. "An Algorithm that Infers Theories from Facts" In Proc. IJCAI-81. Vancouver, Canada, August, 1981, pp. 446-451.
- [17] Yokoi, T., S. Uchida, et al. "Sequential Inference Machine : SIM" In Proc. FGCS-84. Tokyo, Japan, November, 1984, pp. 70-81.