ICOT Technical Memorandum: TM-0091

TM-0091

Source-Level Optimization Techniques for Prolog

Hajime Sawamura, Taku Takeshima
and Akihiko Kato
(Fujitsu Ltd.)

January, 1985

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

SOURCE-LEVEL OPTIMIZATION TECHNIQUES FOR PROLOG

By

Hajime SAWAMURA *
Taku TAKESHIMA **
Akihiko KATO ***

* Research Staff, Fundamental Informatics Section,
** Senior Research Staff, Information Engineering Section,
*** Associate Research Staff, Fundamental Informatics Section,
International Institute for Advanced Study of Social Information Science (IIAS-SIS), Fujitsu Ltd.

Abstract

A source-level optimizer for Prolog, a nondeterministic logic programming language, is designed and implemented for the purpose of Prolog program improvement.

The design principles of the optimizer include the following three phases; (1) information extraction, (2) inline expansion, (3) local optimization. In the phase (1), each predicate is classified into any one of the three types: straight-line, tail-recursive, general recursive, and is decided whether it is deterministic or not, according to the concepts of determinacy: a-determinacy and r-determinacy. In the phase (2), automatic cut insertion is done to avoid unnecessary backtracking, and then the predicates are expanded (partially evaluated) by the inline substitutions of their defining predicates to them. In the phase (3), various local optimization techniques are applied to the resultant predicates of the inline expansions. As the local optimization techniques, are established (a) partial unification, (b) propositional simplifications (deletion of multiple conjuncts and disjuncts, deletion of redundant "true" and "fail" predicates, deletion of unexecutable parts, factoring), (c) deletion of redundant variables by equality substitution, (d) integration of equational predicates, (e) resolution of disjunctive goals, etc.

Some theoretical considerations and results on the optimization of Prolog are added, including the Church-Rosser property of certain optimization techniques and the recursive unsolvability of the determinacy of predicates.

1. Introduction

Program optimization is program transformation by which programs are improved into more efficient ones. It must be examined at the different levels of the language hierarchy, ranging from source to object. The aims and effects of the optimization are different in each level. The merits of optimizing programs at the source-level lie in the mutually relating points that by the source-level optimization, computation can be partially done in advance and various redundancy in a program text can be eliminated in the sense of sumbolic execution.

This paper is concerned with optimizing programs in Prolog [1], a nondeterministic logic programming language, at the source-level. The basic purpose of the optimizing techniques presented is to improve the excution time of programs rather than space effciency. Prolog programs deserves optimization at the source-level since it is a programming language derived from a logic and hence it allows to describe algorithms in a rather descriptive manner, compared with the conventional programming languages. Furthermore, the study of optimizing Prolog would turn out to be useful as an aid for the efficient implementation of nonprocedural languages.

So far, there has been neither so much work as to the source-level optimization for Prolog nor other nondeterministic programming languages except for [2], [3], etc. In [2], Chikayama proposes a brief sketch of ideas on what sort of optimization could be done in source code level of logic programming language programs. In [3], based on unfold/fold transformation techniques

[4], Tamaki and Sato present a heuristical transformation system for pure Prolog.

In this paper, we describe the optimization method for Prolog, without restricting it to pure Prolog, and propose a practically useful optimizer which is the first attempt as far as we know. Our approach differs significantly from the current situation of program transformation [4] as mentioned above because our optimization method may be weak but general, in contrast with strong but specialized program transformation. Our optimization method has been established by the following three design principles:

(1) Information extraction,

(2) Inline expansion,

(3) Local optimization

In optimizing programs, it is convenient to extract information from the program text beforehand. In the present paper, we deal with predicate types and determinacy. Each predicate is classified into any one of the three types: straight-line, tail-recursive, general recursive, and is decided whether with the help of r-determinacy it is a-deterministic or not. These concepts are defined in Section 2.

Before programs are expanded in-line, cuts are inserted into the place where unnecessary backtracking could occurs, so that the optimization of nondeterminacy in terms of backtracking can be partly realized. In general, the main purpose of the inline expansion are twofold: to delete subroutine linkage overhead and to increase opportunities for local optimizations by providig more global program units for them [5]. The inline expansion for Prolog is done in such a way that the predicates(goals) are

expanded (partially evaluated) by the inline substitutions of their defining predicates to them. This must be subject to the execution mechanism of Prolog programs, that is, the leftmost and depth-first execution. Although the inline expansion may tend to make program sizes rather large in general, such an aspect of the inline expansion is left untouched in this paper.

Various local optimization techniques are devised, which may be independently applied to an initial program text, or applied to those predicates which were made longer by the inline expansions. They consists of (a) partial unification, (b) propositional simplifications(deletion of multiple conjuncts and disjuncts, deletion of redundant "true" and "fail" predicates, deletion of unexecutable parts, factoring, etc.), (c) deletion of redundant varibles by equality substitution, (d) integration of equational goals, (e) resolution of disjunctive goals, etc.

The optimizaion techniques in conventional programming languages are classified into two groups: interprocedural technique and intraprocedural one. Thc inline expansion is an interprocedural technique and the various local optimization techniques are intraprocedural one.

We do not intend to provide a formal treatment of source-to-source transformations, nor to prove that certain transformations, under specific semantics, preserve program equivalence. This paper is not concerned with considering the program improvement for Prolog from the complexity-theoretic point of view, neither. Our purpose is, rather, to elucidate or pursue possibilities of natural and intuitionistic optimization techniques for Prolog at the source level. In other words, our main concern lies in presenting some ideas on what sort of

optimization could be done at the source-level of Prolog programs.

The remainder of the paper is divided into 5 sections. Section 2 contains the notations and definitions needed to describe various optimization schema. Sections 3 presents the optimization techniques and the optimization schema together with their application conditions. Section 4 contains some theoretical considerations and results which are useful for validating the implementation issues of the optimizer, as well as the results concerning the recursive unsolvability of the determinacy of predicates. Section 5 describes the outline of the Prolog optimizer implemented in Prolog and some illustrative examples. Finally , discussions and concluding remarks are included in Section 6.

2. Notational conventions and definitions

We assume that the readers are familiar with the syntax and the semantics of Prolog [1]. Here, only the notations and definitions needed to describe the optimization schema, which will be introduced in the succeeding sections, are given.

A Prolog program is a finite set of clauses of the form "H :- B_1, ... , B_n.", where $n \geq 0$. H is called the head and B_i's are calld the goals or predicate calls. Given a goal (a predicate name), a set of clauses whose heads have the same name and arity as the goal (the predicate name) is called a predicate definition of the goal (the predicate name), or the defining clauses of the goal (the predicate name).

It should be noted that we use the distinguished symbols for syntactical variables ranging over the syntactic domains of Prolog.

[Notational conventions]

(1) The letters P, G, H (with or without subscripts) represent goals (predicate calls) or heads, which are of the form of predicate names followed by some arguments, letters X, Y, Z (with or without subscripts) represent variables, letters t, s (with or without subscripts) represent terms of Prolog, and letters p,q,r represent predicate names or propositions.

(2) The boldface letters **S** and **T** (with or without subscripts) represents (possibly empty) sequences of goals which are delimited by commas. If **S** is an empty sequence, it denotes "true" predicate. A boldface letter **A** (with or without subscripts) represents a nonempty sequence of terms in Prolog.

(3) If **S**(X) is a sequence of goals in which a variable X may occur, then **S**(t) represents a sequence of goals obtained by substituting a term t for every occurrences of a varible X in **S**(X).

(4) The boldface letters **P** and **Q** represent vertical rows of goals and clauses.

(5) If t is a term of Prolog, then t' is a term obtained by renaming all variables in t.

In order to illustrate that a sequence of goals (or a clause) is transformed into an optimized one possibly by using appropriate predicate definitions, we use a horizontal line which corresponds to derivability in logic.

[Optimization schema]

The optimization schema is a figure of the form

P

Q

where **P** and **Q** are called an upper sequence and a lower sequence of the optimization schema respectively.

In our optimizer, predicate definitions are classified into three types: straight-line, tail-recursive, general recursive. According to them, programs are expanded in-line.

[Types of predicate definitions]

(1) A clause H :- **S** is said to be a straight-line clause if there is no predicate call with the same predicate name as that of the head H in **S**.

(2) A predicate definition of a goal, **P**, is said to be straight-line if all the clauses of **P** are straight-line clauses.

(3) Let a predicate definition of a predicate name p, **P**, be

H_1 :- $\mathbf{S}_1$, P_1.

.

.

.

H_n :- $\mathbf{S}_n$, P_n.

where,

(i) there is no predicate call with name p in $\mathbf{S}_1$, ... , $\mathbf{S}_n$ and,

(ii) P_1, ... , P_n are atomic predicates and there exists at least one predicate call with name p among P_1, ... , P_n.

Then, the predicate definition **P** is said to be tail-recursive .

(In our optimizer the restriction (ii) is actually weakened, so that P_i may be, for example, of the form "G_1 ; G_2" provided that "P_i :- G_1." and "P_i :- G_2." are tail-recursive.)

(4) If a predicate definition is neither straight-line nor tail-recursive, it is said to be general recursive.

The determinacy of a predicate plays very important roles in optimizing nondeterministic programs; actually in expanding a predicate call by its defining clauses with cuts and in inserting cuts into the place where unnecessary backtracking may occur. The exposition will be described in the next section.

Our definition of the determinacy is as follows.

[Determinacy of a goal (a predicate call)]

A goal (or a predicate call) is deterministic if when it is called at most one clause of its defining clauses succeeds, and when it is backtracked it never succeeds again.

In Section 4, it is shown that it is undecidable whether for any predicate it is deterministic or not. Therefore, the concepts of algorithmically decidable deterministic predicates must be introduced.

We need the two kinds of concepts of determinacy which are mutually defined. In the following definition, we assume that the constructs dynamically modifying a program, such as "assert", "retract" etc., do not appear in the program.

[a-determinacy and r-determinacy]

A predicate call p(**A**) is termed a-deterministic or r-deterministic if it satisfies the following mutually recursive

conditions.

(i) If p is a built-in (evaluable) predicate of Prolog and p(**A**) is deterministic, then p(**A**) is a-deterministic and r-deterministic.

(ii) Let a predicate definition **P** of the predicate name p be

H_1 :- $\mathbf{S}_1$.

.

.

.

H_i :- $\mathbf{S}_{i1}$, [!,] $\mathbf{S}_{i2}$., where the cut symbol "!" (if any) is rightmost.

.

.

.

H_n :- $\mathbf{S}_n$.

Then, for each i ($1 \leq i \leq n$), if either (1) or (3) of the following conditions holds, then p(**A**) is a-deterministic, and for each H_i which is unifiable with p(**A**), if (1), (2) or (3) holds, then p(**A**) is r-deterministic:

(1) There is no cut symbol in the body of the i-th clause, it is the last clause in the predicate definition **P**, and every goal of $\mathbf{S}_{i1}$ and $\mathbf{S}_{i2}$ is a-deterministic or r-deterministic.

(2) There is no cut symbols in the body of the i-th clause, p(**A**) is not unifiable with any H_j ($i+1 \leq j \leq n$) and every goal of $\mathbf{S}_{i1}$ and $\mathbf{S}_{i2}$ is a-deterministic or r-deterministic.

(3) There exist cut symbols in the body of the i-th clause and every goal of $\mathbf{S}_{i2}$ is a-deterministic or r-deterministic.

It can be easily checked that if a predicate p(**A**) is a-determinisic or r-deterministic, then it is deterministic.

The definitions of a-determinacy and r-determinacy have been given, depending on the three concepts: cut's behavior, deterministic built-in predicates and unifiability statically determined. In other words, they never refer to what types of arguments a predicate takes when it is called. These two concepts, a-determinacy and r-determinacy seem to be less complicated and better concepts than other computer-checkable determinacy in the sense that they can be determined without committing the semantics of a predicate. Here, by the semantics of a predicate we mean to prescribe the domain of terms in which the predicate succeeds. However, prescribing such a semantics for a predicate beforehand would be obviously impossible.

Note that if a predicate call is a-deterministic, it is always deterministic without depending on its argument form. In other words, an a-deterministic goal is absolutely deterministic in the sense that it does not depend on its argument form in the goal. Therefore, when a goal p(**A**) is found to be a-deterministic, we sometimes call the predicate p a-deterministic or simply deterministic. In contrast to the absolute determinacy of a-determinacy, an r-deterministic predicate call is relatively deterministic since its predicate depends on how it is called. Furthermore, in our definition, a deterministic predicate except for built-in predicates can not be determined to be a-deterministic if the number of its defining predicates is more than 2 and there exist no cut symbols in them. At the propositional level, that is, when the predicate to be examined is a proposition, a-determinacy coincides with r-determinacy.

Example 1. The predicate p is a-deterministic.

```
p(a) :- write(a1), nl, !, write(a2).
p(b) :- write(b).
```

Example 2. The goal q([a,b]) is r-deterministic, but the goal q(X) is not.

```
q([c]) :- write(c), p(X).
q([a|X]) :- write(a), p(X).
```

where the predicate call p(X) calls its defining clauses given in Example 1.

3. Optimization techniques for Prolog programs

Various optimization techniques and optimization schema together with their application conditions are presented in this section.

3. 1 Partial unification

In conventional programming languages, so-called calling mechanism [5] is conveniently eliminated by soubroutine expansion technique. In Prolog, however, a large part of calling mechanism is often remained as an equational goal which expresses unifiability of a goal with a head of its defining clauses. Partial unification reduces the process of runtime unification to some extent. Furthermore, in our optimizer, the results of partial unification, a sequence of equational goals, are used in the further stages of the successive optimization process, such as the equality substitution described in the subsection 3.3.7. And finally, the remaining equational goals turns out to be

integrated into one equational goal by using the optimization schema: integration of goals described in the subsection 3.3.8.

[Optimization schema 1] (partial unification)

An equational goal, which expresses unifiablity of the two terms of the both hand sides in Prolog, is transformed into a sequence of equalities in a solved form [6] if it exists, otherwise, simply a goal "fail".

$$\frac{f(\ldots) = f(\ldots)}{X_1 = T_1, \ \ldots \ , X_n = T_n} \quad \text{or} \quad \frac{f(\ldots) = f(\ldots)}{\text{fail}}$$

where $X_1, \ \ldots \ , X_n$ are variables occurring in the upper equality and $T_1, \ \ldots \ , T_n$ are terms.

Example 3.

$$\frac{p(X,g(Y,X,c)) = p(h(Z),g(Z,h(d),c))}{X = h(d), \ Y = d, \ Z = d}$$

3. 2 Automatic insertion of cut symbols

Cuts should be inserted into the place where unnecessary redo could occur on backtracking, so that the optimization of nondeterministic programs based on backtracking can be partly realized. We accomplish this in the following case.

[Optimization schema 2]

H_1 :- S_1.

.

.

.

H_i :- S_{i1}, S_{i2}.

.

.

.

H_n :- S_n.

H_1 :- S_1.

.

.

.

Hi :- S_{i1}, !, S_{i2}.

.

.

.

H_n :- S_n.

where either of the following conditions is satisfied:

(i) If there exists no cut in S_{i1}, then the i-th clause is a last clause of the program and every predicate in S_{i1} is a-deterministic or r-deterministic,

(ii) if there exist cuts in S_{i1}, then no cuts occur in S_{i2}, and every predicate in S_{i1}, occurring in the right side of the rightmost cut symbol in S_{i1} is a-deterministic or r-deterministic.

Example 4.

```
r([c]) :- write(c), !, p(X).
r([a|X]) :- write(a), p(X).
```

```
r([c]) :- write(c), !, p(X), !.
r([a|X]) :- write(a), p(X), !.
```

where the predicate call p(X) calls its defining clauses given in Example 1.

3. 3 Inline expansion

A Prolog programs has, by nature, several alternative clauses for a predicate. Due to this nondeterminancy of Prolog, inline expansion techniques is rather complicated in Prolog than ordinary programming languages. Here, a natural method for inline expansion of Prolog programs is presented. That is, " a predicate call is replaced by a disjunction of alternative clauses of its defining clauses, each preceded by a sequence of equational goals which represents the unifiability of the call with a head of its defining clauses. "

It is noted, however, that this replacement is valid only when none of alternative clauses has cuts in its body. Because, the cuts brought into the original clause cause a different control flow in general. The next example shows that difference.

(a) Before inline expansion:

```
p :- q, a, r.
p.
a :- b, !, c.
a :- d.
```

(b) After inline expansion:

```
p :- q, (a = a, b, !, c ; a = a, d), r.
p.
a :- b, !, c.
a :- d.
```

In the program (a), suppose the predicate call c fails. Then, the predicate call a fails and the control backtracks to q. On the other hand, the failure of the predicate call c in the program (b) causes the predicate call p to fail.

Thus, the existence of cut symbols in the called predicate have a serious influence upon the possibilities of inline expansion. In what follows, the methods of inline expansion are formally introduced, together with the conditions which allow to expand a predicate call by using its defining clauses which include cuts.

In this paper, a clause with no body, "P.", is identified with a clause "P :- **S**.", where **S** is empty, consequently "P :- true."

3. 3. 1 Inline expansion by programs with no cut

An goal G is expanded by its defining clauses with no cut as follows.

[Optimization schema 3]

G.

H_1 :- $\mathbf{S}_1$.

.

.

.

H_n :- $\mathbf{S}_n$.

$G = H_1', \mathbf{S}_1'$; ... ; $G = H_n', \mathbf{S}_n'$

H_1 :- $\mathbf{S}_1$.

.

.

.

H_n :- S_n.

where S_i ($1 \leq i < n$) does not contain any cut, H_i ($1 \leq i \leq n$) is a defining clause of G, and G = H_i', S_i' simply denotes G = H_i' if S_i is empty.

Example 5.

```
append([a,b],Y,Z).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
append([],L,L).
```

```
append([a,b],Y,Z) = append([_X|_L1],_L2,[_X|_L3]),
append(_L1,_L2,_L3) ; append([a,b],Y,Z) = append([],_L,_L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
append([],L,L).
```

3. 3. 2 Inline expansion by programs with cuts

(1) Cut schema (case 1)

We consider here such a special case that each of the defining clauses which are used for inline expansion includes just one cut symbol in it.

Let such a defining clauses be

H_1 :- S_{11}, !, S_{12}.

.

.

.

H_n :- S_{n1}, !, S_{n2}. | H_n :- S_n. | H_n. ("|" denotes "or")

where S_{i1}, S_{i2} ($1 \leq i \leq n$) and S_n does not contain any cut symbol,

and if S_{i1} or S_{i2} is empty, the body " S_{i1}, !, S_{i2} " denotes

" ! ", when both S_{i1} and S_{i2} are empty,

" !, S_{i2} ", when only S_{i1} is empty, or

" S_{i1}, ! ", when only S_{i2} is empty.

Then, a goal G is expanded in-line, following the schema.

[Optimization schema 4]

G.

H_1 :- S_{11}, !, S_{i2}.

.

.

.

H_n :- S_{n1}, !, S_{n2}. | H_n :- S_n. | H_n.

G = H_1', S_{11}' -> S_{12}' ; ... ; G = H_n', Sn_1, -> S_{n2}' | G = H_n', S_n' | G = H_n'

H_1 :- S_{i1}, !, S_{i2}.

.

.

.

H_n :- S_{n1}, !, S_{n2}. | H_n :- S_n. | H_n.

where if S_{i1} or S_{i2} is empty, the i-th disjunct of the lower sequence is constructed as follows:

G = H_i' -> S_{i2}', when the i-th clause is H_i :- !, S_{i2}.,

G = H_i', S_{i1} -> true, when it is H_i :- S_{i1},!., or

G = H_i' -> true, when it is H_i :- !.

Example 6.

transform(T).

```
transform('end_of_file') :- !.
transform(T) :- fold(T), write(R), Write('.'), nl, !, fail.
```

```
transform(T) = transform('end_of_file') -> true ;
transform(T) = transform(_T), fold(_T), write(_R),
write('.'), nl -> fail.
transform('end_of_file') :- !.
transform(T) :- fold(T), write(R), Write('.'), nl, !, fail.
```

(2) Cut schema (case 2)

Next, we consider such a case that the determinacy enables to expand programs in-line even if defining clauses contain cuts.

[Optimization schema 5]

H_1 :- S_1.
.
.
.
H_i :- S_{i1}, p(A), S_{i2}.
.
.
.
H_n :- S_n.
$p(A_1)$:- T_1., where cuts appear in some clause of the
. defining clauses of the predicate name p.
.
.
$p(A_m)$:- T_m.

H_1 :- S_1.

.

.

.

H_i :- S_{i1},
(p(A) = p(A_1)', T_1' ; ... ; p(A) = p(A_m)', T_m'),
S_{i2}.

.

.

.

H_n :- S_n.

p(A_1) :- T_1.

.

.

.

p(A_m) :- T_m.

where either of the following conditions is satisfied:

(1) If there exists no cut symbol in S_{i1}, then the i-th clause is the last clause of the clauses with the heads H_j's ($1 \leq j \leq i$) and every predicate in S_{i1} is a-deterministic or r-deterministic,

(2) if there exist cut symbols in S_{i1}, then every predicate call in S_{i1} which appears on the right hand side of the rightmost cut symbol in S_{i1} is a-deterministic or r-deterministic.

Note that even if cuts appear in the bodies T_i's ($1 \leq i \leq m$), the above expansion can hold without the expansion conditions (1) and (2) if the unifiability of the predicate call p(A) with any head p(Ai) including cuts is known to fail. However, currently we are not concerned with these situations for simplicity.

Example 7.

```
r(a,Y,Z) :- !, q(Y), append(a,Y,Z).
r(b,Y,Z) :- p(b), append(b,Y,Z).
append([],L,L) :- !.
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3),!.
```

```
r(a,Y,Z) :- !, q(Y),append(a,Y,Z).
r(b,Y,Z) :- p(b), (append(b,Y,Z) = append([],_L,_L), ! ;
             append(b,Y,Z) = append([_X|_L1],_L2,[_X|_L3]),
             append(_L1,_L2,_L3),!).
append([],L,L) :- !.
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3),!.
```

where the predicate calls q([a,b]) and p(b) call the defining clauses given in Example 1 and 2 respectively.

In the subsections below, the methods of inline expansion described up to now are incorporated into the methods of expanding a program in which various types of predicates are defined. Although predicate definitions are expanded in-line according to their types, our stepwise expansion method essentially amounts to expanding the goals which call straight-line predicate definitions, except that the tail-recursive predicate calls can be expanded once only. This is due to the current research situation that the objective expansion criterion for recursive Prolog programs have not been found yet. Therefore, it must be noted that the overall expansion method described below is not our final solution to the inline expansion. Instead, in this paper, as a first step towards an

ideal inline expansion method we employ an intermediate step in which the original types of predicate definitions are preserved by the inline expansions. This is because the type-preserving expansions are convenient to assure program reliability and also to debug. In fact, preserving the overall structure of a program does not so much destroy the intention of a programmer.

In what follows, the emphasis is placed on the conditions for the inline expansions rather than the algorithms of them. The application of Optimization schema 4 is not currently considered in the description below.

3. 3. 3 Inline expansion of straight-line predicate definitions

Suppose we have a program which consists of various predicate definitions. Each goal of each clause in a straight-line predicate definition of a predicate name is expanded in-line, using the corresponding predicate definitions in the program. This is done in a top-down, leftmost and depth-first execution fashion of Prolog, as follows.

Choose a predicate definition of a predicate name p.

(i) Assume that the predicate p is a-deterministic. Let its predicate definition be

H_1 :- $\mathbf{S}_1$.

.

.

.

H_i :- $\mathbf{S}_i$,[!], $\mathbf{T}_i$., where the cut symbol "!" (if any) is

. rightmost.

.

.

H_n :- S_n.

Then,

for each goal in S_i :

(1) if the goal satisfies the following conditions, expand it by means of Optimization schema 3;

(a) no cut appears in the predicate definition of the goal,

(b) the defining clauses of the goal to be expanded is straight-line,

(c) in the disjunctive term which is generated by expanding each goal, no predicate name appears which have been already expanded, including the predicate name p,

and then for each goal of the resulting disjunctive term (if any), repeat this,

(2) if cuts are included in the defining clauses of the goal, then expand it by means of Optimization schema 5, and then expand each goal in the disjunctive term generated, in the same manner as (1) above,

for each goal in T_i and each goal in the last clause :

(3) if it satisfies the conditions (b) and (c), then expand it by means of Optimization schema 3, and then expand each goal in the disjunctive term generated, in the same manner as (1) above.

(ii) Otherwise, expand each goal in the defining clauses of the predicate p, in the same manner as (1) or (2) above.

Note that:

(1) The condition (c) above is a termination condition for the inline expansion of a straight-line predicate definition, which prohibits a circular expansion.

(2) The result of the inline expansion of a straight-line predicate definition is straight-line again with the help of the condition (b).

Example 8. Consider the following program:

$p:-p_1,q,p_2.$

$q:-q_1,r,q_2.$

$r:-r_1,p,r_2.$

Each clause is straight-line. The goal q in the first clause is expanded by the second clause, resulting $p:-p_1,(q=q,q_1,r,q_2),p_2.$ This is not expanded any more. The second clause is expanded by the third clause, resulting $q:-q_1,(r=r,r_1,p,r_2),q_2.$, which is in turn expanded to $q:-q_1,r=r,r_1,(p=p,p_1,q_1,r,q_2,p_2),q_2.$, by using the new clause for p. Finally, the third clause is left unexpanded.

3.3.4 Inline expansion of tail-recursive predicate definitions

It is often done to convert tail-recursive programs to iterative programs as in Prolog compiler. The characteristics of Prolog programs, however, suggests that this transformation is not meaningful in optimizing Prolog programs at the source-level.

In what follows, is presented such a method that iteratively accomplished goals in a recursive clause are expanded in-line. This is the same idea as the subroutine expansion in a loop of iterative programming languages and the effect of the inline expansion is most expected. Furthermore, since it is observed

that Prolog programs are almost of recursive form, especially of tail-recurvive form, the effect of such an inline expansion method is considered extremely large.

After having expanded straight-line predicate definitions , tail-recursive predicate definitions are expanded in-line.

(i) All the straight-line clauses occurring in a tail-recursive predicate definition are expanded in-line in the same manner as the subsection 3.3.3.

(ii) Let one of the defining clauses of a predicate name p be

p(**A**):-**S**,p(**B**).

where there is no call of a predicate p in **S**. Then, each goal G in **S** is expanded in-line in the same manner as the subsection 3.3.3.

(iii) The tail-recursive goal p(**B**) is expanded once only by using itself, that is, the currently constructed definition of the predicate p, as follows.

(1) If the predicate p is a-deterministic, then expand the tail-recursive goal once only by means of Optimization schema 3.

(2) If the predicate definition of the predicate p includes cuts, then expand the tail-recursive goal once only by means of Optimization schema 5.

Note that:

(1) The termination condition of the inline expansion of straight-line predicate definitions gives that of tail-recursive predicate definitions as well.

(2) The tail-recursiveness of a predicate definition is preserved by the inline expansion, in the sense stated in Section 2.

3.3.5 Inline expansion of general recursive predicate definitions

Each non-recursive goal of each clause in a general recursive predicate definition is expanded in-line in the same manner as the goal expansion of straght-line predicate definitions described in the subsection 3.3.3. Note that the general recursiveness of a predicate definition is preserved as well.

3.4 Simplification of a sequence of goals

In this subsection, various techniques for simplifying a sequence of goals at the propositional level are presented. Most of them are local simlification rules or deletion strategies and are often applied to the resulting clauses after inline expansion as well as used individually within a clause.

3.4.1 Deletion of multiple occurrences of goals

(i) Deletion of multiple conjuncts

Any identical goal occurring in a sequence of conjunctive goals is deleted except for the leftmost goal, by the repeated applications of the following schema.

[Optimization schema 6]

$$\frac{S_1, P, S_2, P, S_3}{S_1, P, S_2, S_3}$$

where the goal P in the lower sequence is the leftmost occurrence of it and the following conditions must be satisfied:

(1) P is a-deterministic.

(2) P is not a predicate call with side effect such as built-

in input/output predicate or a meta predicate, and furthermore it is not extra control predicate such as cut symbol "!", "repeat".

(3) "not" predicate does not occur in the upper sequence.

Here we list some counterexamples of the optimization schema in which the conditions above are violated.

(a) p, q, p

---

q,p

is not correct. For, consider non-terminating q in the upper sequence but terminating q in the lower sequence.

(b) S_1, var(X), p(X), var(X), S_2

S_1, var(X), p(X), S_2

is not correct since "var"is a meta predicate.

(c) S_1, write(X), p(X), write(X), S_2

S_1, write(X), p(X), S_2

is not correct since "write" is an output predicate.

(d) Suppose we have the following assertions:

q(a).

q(b).

q(c).

Then,

repeat, q(X), repeat, not(X = a)

---

repeat, q(X), not(X = a)

is not correct since in the upper sequence the first success of q(X) with X = a forces to repeat "not(X = a)" indefinitely, but in the lower sequence the second success of q(X) with X = b

completes the execution.

(e) Suppose we have the assertions:

```
p(f(Y,V)).
p(f(b,c)).
p(f(b,d)).
q(f(b,U)).
q(f(e,V)).
```

Then,

```
p(X), q(X), p(X), not(X = f(b,c))
-----------------------------------
p(X), q(X), not(X = f(b,c))
```

is not correct since the upper sequence terminates with X = f(b,d), but the lower one terminates with X = f(e,V).

(ii) Deletion of multiple disjuncts

Any identical goal occurring in a sequence of disjunctive goals is deleted except for the leftmost goal.

[Optimization schema 7]

S_1; P; S_2; P; S_3

S_1; P; S_2; S_3

where the goal P in the lower sequence is its leftmost occurrence and the folowing conditions must be satiafied:

(1) P is not a predicate call with side effect.

(2) Backtracking control never enters to the part "P; S_2; P" in the upper sequence.

The following exemplifies the significance of the condition (2):

(p; q; p), write(a), fail

(p; q), write(a), fail

3.4.2 Deletion of redundant "true" and "fail" predicates

Redundant "true" (including a goal of the form X = X) and "fail" (sometimes expressing a failure of unification) predicates can be deleted without any condition.

[Optimization schema 8-1]

S_1, true, S_2

S_1, S_2

[Optimization schema 8-2]

S_1; fail; S_2

S_1; S_2

3.4.3 Deletion of unexecutable parts

The sequence of goals to which control never reaches is removed from the program text.

[Optimization schema 9]

S_1, fail, S_2

S_1, fail

Note, however, that the following dual schema is not valid:

S_1; true; S_2

S_1; true

3.4.4 Factoring of common goals

A sequence of goals is factorized for both the efficiency and clarity of programs. The factoring of common goals is similar to the inverse of distribution rule over formulas.

[Optimization schema 10]

S, S_1, **T**; **S**, S_2, **T**

S, (**S**$_1$; **S**$_2$), **T**

where any predicate in **S** and **T** is side effect free.

Example 9.

p, q; p, q, r

---

p, q, (true; r)

3.5 Deletion of redundant variables

In this subsection, a method of deleting redundant Prolog variables is presented. This is suggested by equality substitution rule in first-order logic and is often applied to the resulting clauses after the inline expansion as well as used individually within a clause. This optimization technique is also useful in reducing the number of goals to be accomplished.

[Optimization schema 11-1] (equality substitution)

p(..., X ,...) :- **S**, X = t, **T**.

---

p(..., t ,...) :- **S**(t), **T**(t).

where any of predicates with side effect, a cut symbol and meta predicates does not appear in **S**.

Note that this optimization schema includes such a special case with no effect that

p(Y) :- q(Y), X = t, r(Y).

---

p(Y) :- q(Y), r(Y).

and implies the following optimization schema as well

P :- **S**, X = f(t_1, ... ,t_n), X = f(r_1, ... ,r_n), **T**.

P :- **S**, f(t_1, ... ,t_n) = f(r_1, ... ,r_n), **T**.

With violating the conditions above,

(a) q(X) :- !, X = t, p(X).

---

q(t) :- !, p(t).

is not correct if there exist successfull alternatives w.r.t. the predicate name q.

(b) p(X) :- write(X), X = a.

---

p(a) :- write(a).

is not correct since both the upper and lower sequences fail when called by p(b) for example, but the upper sequence produces b as the side effect.

[Optimization schema 11-2]

p(..., X ,...) :- $\mathbf{S}_1$; **S**, Y = t, **T**; $\mathbf{S}_2$.

$p(\ldots, X, \ldots) :- S_1;\ S(t),\ T(t);\ S_2.$

with the same proviso as the above optimization schema.

3.6 Integration of goals

In the inline expansion, a calling goal in a clause is replaced by its defining clauses to be needed to accomplish it. Then, the calling mechanism was expressed in terms of a sequence of equational goals which was obtained by the partial evaluation of unification. These equational goals turn out to be partially used in the subsequent local optimization technique such as the equality sybstitution.

In this subsection, the integration of equational goals is presented, where equational goals which was found not to be used any more are integrated into one equational goal. This optimization technique is useful in reducing the number of goals to be acomplished as well as the equality substitution.

[Optimization schema 12]

$$\frac{X1 = t1, \ldots, Xn = tn}{f(X1, \ldots, Xn) = f(t1, \ldots, tn)}$$

where f is an appropriate function symbol.

3.7 Decomposition of a clause

A clause expanded by the inline expansion in the subsection 3.3 is of the form

$$P :- S,\ (S_1;\ \ldots\ ;S_n),\ T.$$

The decomposition of such a clause often proceeds to further improve the resultant clauses, although it seems to be an inverse

transformation of the inline expansion in its form.

[Optimization schema 13]

P :- **S**, (S_1; ... ;S_n), **T**.

P :- **S**, S_1, **T**.

.

.

.

P :- **S**, S_n, **T**.

Note that this optimization schema is valid without any condition , contrary to the inline expansion and is unconditionally useful if **S** is empty or any two heads of the clauses in the lower are made ununifiable by the succeeding optimization techniques.

From another point of view, this optimization schema amounts to enemerate every nondeterministic computation paths as defining clauses.

4. Some theoretical considerations

In this section, some theoretical considerations on the Prolog optimization techniques are summarized, which have been useful for implememting the reliable Prolog optimizer. And the unsolvability of determinacy of goals are described.

(1) Invariant properties

The following two propositions are easily checked from the methods of the inline expansions described in Section 3. That is,

Proposition 1 is an immediate consequence of our inline expansion methods and such an invariance has given an criterion of expanding predicate definitions in-line.

Proposition 1. The types of predicate definitions are invariant through the inline expansions. That is, the straight-lineness, tail-recursiveness and general recursiveness are preserved by the inline expansions.

Proposition 2. a-determinacy is preserved by the inline expansions.

(2) Termination of the inline expansions

The inline expansions have been done in such a way that each goals of a predicate definition is symbolically expanded (partially evaluated) in accordance with Prolog execution order as far as the expansion conditions are satisfied. These are guaranteed to terminate by the following proposition.

Proposition 3. The inline expansions terminate.

Outline of the proof: It is sufficient to show the termination of the inline expansion of a straight-line predicate definition. It is established by the expansion condition (c) given in the subsection 3.3.3, which says that a goal which once have been expanded should not be expanded any more.

(3) Application order of local optimization techniques

Application order of several optimization techniques turns out to be significant when some of them are applicable to the programs expanded by the inline expansions [7]. The fully

optimized programs could not be generated if the application order were not adequate for the source or intermediate programs.

In our optimizer, it is dt_1 (Optimization schema 8-1), dt_2 (Optimization schema 8-2) and du (Optimization schema 9) below which are applied at the intermediate stages of the inline expansions. The others are often applied to the resulting programs after each inline expansion, and their application order can be conveniently determined from their nature, except for Optimization Scheme 13. In this paper, we deal with an analysis of the application order of dt_1, dt_2 and du.

$$dt_1: \frac{S_1, true, S_2}{S_1, S_2}, \quad dt_2: \frac{S_1; fail; S_2}{S_1; S_2}$$

where either S_1 or S_2 is not empty.

$$du: \frac{S_1, fail, S_2}{S_1, fail}$$

where S_2 is not empty.

dt denotes either of the rules dt1 and dt2, and the relation dt-> defined below denotes the relation dt_1 # dt_2, where # denotes the union of relations.

Definition 1. S_1 dt-> S_2 (S_1 du-> S_2) iff S_1 is an upper sequence and S_2 is an lower sequence of the optimization schema dt (du).

Definition 2. A relation * is defined to be $(dt\text{->} \ \#\ du\text{->})^+$, the transitive closure of the union of the relations dt-> and du->.

Definition 3. A sequence **S** is said to be in normal form with

respect to dt and du iff neither dt nor du is applicable to **S**. A sequence **S** is in normal form w.r.t dt (du) iff dt (du) is applicable to **S** no more.

Definition 4. It is denoted by writing dt(**S**) (du(**S**)) that dt (du) is applied to a sequence **S**.

Theorem 4. Let $\mathbf{S}_1$ be a sequence, and $\mathbf{S}_2$ a sequence in normal form. Then, $\mathbf{S}_1 * \mathbf{S}_2$ iff $\mathbf{S}_1 ** \mathbf{S}_2$, where ** is a relation,

$$\#_{m>0,n>0}(\text{du}\rightarrow)^m(\text{dt}\rightarrow)^n,$$

where $(\text{du}\rightarrow)^m$ and $(\text{dt}\rightarrow)^n$ denote m-fold and n-fold product of the relations du-> and dt-> respectively.

Proof.

(only if part) Let $\mathbf{S}_1$ and $\mathbf{S}_2$ be sequences such that $\mathbf{S}_1 * \mathbf{S}_2$. It is not the case that du becomes applicable only after the applications of dt. Therefore, without losing generality, we can apply du to the sequence $\mathbf{S}_1$ at most N times, where N is a total number of fail's occurring in the conjunctive goals of $\mathbf{S}_1$, and next we can apply dt to the resultant sequence at most M times, where M is a total number of true and fail predicates occurring in its sequence, finally obtaining a normal form $\mathbf{S}_2$. From these observations it is claimed that there exist the numbers N and M such that $(\text{du}\rightarrow)^N(\text{dt}\rightarrow)^M$.

(if part) obvious.

Theorem 4 says that in order to reduce a sequence, it is sufficient to apply du several times and then dt several times.

Proposition 5. The normal form w.r.t. du is uniquely determined.

Proposition 6. The normal form w.r.t. dt is uniquely determined.

Theorem 7. The normal form w.r.t. du and dt is uniquely determined.

Proof. By Theorem 4, Proposition 5 and Proposition 6.

Theorem 8. For any sequence, the relation * satisfies Church-Rosser property.

Proof. By Theorem 4 and Theorem 7.

Based on these results, the program deleting redundant true and fail predicates, and the program deleting unexecutable goals from a sequence of goals are implemented as $(dt)^n$ $(n \geq 0)$ and $(du)^m$ $(m \geq 0)$ respectively, and for any sequence, $(du)_m$ is first applied to it and then $(dt)_n$ is applied.

(4) Recursive unsolvability of determinacy

The concept of determinacy of a predicate has played important roles in our optimization schemas and in Section 2 it is claimed, however, that the determinacy of a predicate is undecidable. Here, the proof of the recursive unsolvability of determinacy is given. And its implications are then examined.

Let us recall the definition of a deterministic predicate.

Definition 5. A goal (or a predicate call) is deterministic if when it is called at most one clause of its defining clauses succeeds, and when it is backtracked it never succeeds again.

Note that with this definition, a predicate call which does not terminate at the first execution is deterministic, and a predicate call which succeeds at the first execution but does not terminate on backtracking is deterministic as well.

Before going into a proof of the recursive unsolvability of

the determinacy, it must be noted that computable functions are computable in Prolog.

Theorem 9. No algorithm exists for deciding whether for any predicate call it is deterministic or not.

Proof. Suppose there exists an algorithm which realizes a predicate det: for any predicate call P

det(P) = success, if P is deterministic,
failure, otherwise.

Here, consider the following program:

```
q :- det(q).
q.
```

For this program,

(i) Suppose det(q) = success. Then on backtracking, a call q succeeds again, or else it succeeds in its second clause. Therfore it is not deterministic.

(ii) Suppose det(q) = failure. Then a call q succeeds only in its second clause. Therfore it is deterministic.

Both cases lead to contradictions. Consequently such an algorithm det does not exist.

It should be remarked that :

(1) In the program of the predicate q, if the first clause is changed into a clause "q :- det(q), !.", the above case (i) does not lead to a contradiction, however in the proof it has been shown that the existence hypothesis allows us to make such a curious program q that raises contradictions.

(2) Our argument in the proof can be also applied to the case that the predicate det is written explicitly as a two-place

predicate such as det(P, Defs), where Defs is a set of defining clauses to be needed for deciding the determinacy of a predicate call P. In the proof we let the predicate det be a unary predicate, for clarity.

Corollary 10. No algorithm exists to decide whether for any predicate call it is nondeterministic or not.

Proof. Obviously, the existence of such an algorithm implies that of an algorithm deciding determinacy, which contradicts Theorem 10.

Corollary 11. No algorithm exists which answers the number of the solutions of any predicate call.

Proof. Such an algorithm turns out to answer the number of the solutions of a deterministic predicate as a special case. but it is impossible by Theorem 10.

Corollary 12. No algorithm exists for deciding whether for any proposition (without any variable) it is deterministic or not.

Proof. The proof of Theorem 10 can be restated by using "any proposition p" instead of "any predicate call P".

This corollary says that even at the propositional level, deciding the determinacy of a predicate call is impossible in principle.

Corollary 13. Suppose that an algorithm of the following predicate det# exists: for any terminating predicate call P,

det#(P) = success, if P is deterministic,
failure, otherwise.

Then, there exists a nonterminating predicate call r such that det#(r) does not terminate.

Proof. Consider the following program:

```
r :- det#(r).
r.
```

A predicate call terminates or does not terminate. Suppose the predicate call r terminates. Then, the same contradictions as those in the proof for Theorem 10 arise. Theorefore the predicate call r does not terminate. This implies that det#(r) does not terminate, according to the definition of the predicate r.

5. Outline of an implementation and illustrative examples

5. 1 Outline of an implementation

An experimental system for Prolog program improvement, a Prolog-optimizer, has been implemented on the computer Dec2060 and VAX/UNIX, written in Dec10 PROLOG and C-PROLOG respectively. Various optimization schemas described in the former sections were integrated into a Prolog-optimizer, takimg into account some theoretical considerations described in Section 4. In the present implementation, the users must indicate an application order of various local optimization schemas, at the place where the order can not be determined uniquely and adequtely. In the future, a better application order would be settled through a number of practical example programs. Fig 1. illustrates the global control flow of the system.

The system basicallly consists of three components:

(1) Input-output routines,

(2) Information extracting routines,

(3) Optimizing routines.

Input-output routines read in a source program from an input file and write out the resultant improvd program to an output file. Information extracting routines realize the clasification of predicate types and the detection of determinism. Optimizing routines realize various optimization techniques described in Section 3.

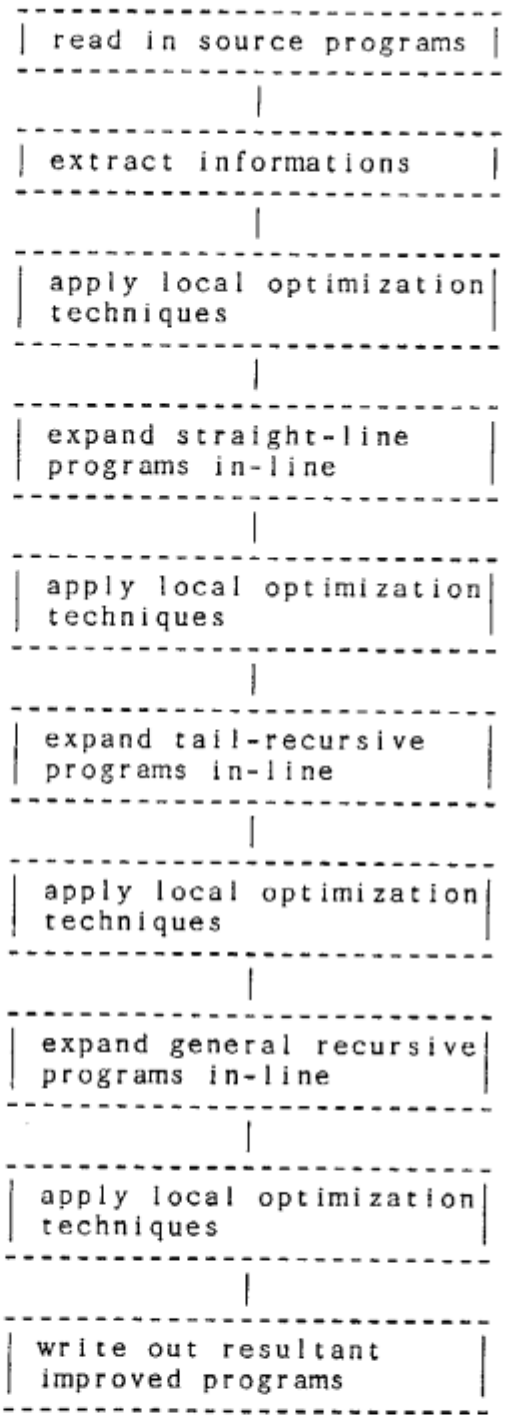

Fig. 1 Control flow of the Prolog optimizer

5. 2 Illustrative examples

In this section, some practical examples are described,

illustrating the overall control flow of processing source programs by the Prolog optimizer.

Example 10.

As a first example, we take an well-known "reverse" program of a list, which serves as an illustration of the overall integrated procedure of various optimization schemas. In its definition, r(X, Y, Z) reads as Y is a list which is resulted in appending the reverse of X to Z.

It is successively transformed as follows:

```
reverse(X, Y) :- r(X, Y, []),!.
r([], Z, Z).
r([H|T], W, Z) :- r(T, W, [H|Z]).
```

```
reverse(X, Y) :- r(X, Y, []),!.
r([], Z, Z).
r([H|T], W, Z) :- r(T, W, [H|Z]) = r([], Z1, Z1) ;
      r(T, W, [H|Z]) = r([H1|T1], W1, Z1), r(T1, W1, [H1|Z1]).
      (by inline expansion)
```

```
reverse(X, Y) :- r(X, Y, []),!.
r([], Z, Z).
r([H|T], W, Z) :- T = [], W = Z1, Z1 = [H|Z] ;
      T = [H1|T1], W = W1, Z1 = [H|Z], r(T1, W1, [H1|Z1]).
      (by partial unification)
```

```
reverse(X, Y) :- r(X, Y, []),!.
r([], Z, Z).
```

```
r([H|T], W, Z) :- T = [], W = Z1, Z1 = [H|Z].
r([H|T], W, Z) :- T = [H1|T1], W = W1, Z1 = [H|Z],
                  r(T1, W1, [H1|Z1]).
```

(by decomposition of a clause)

```
reverse(X, Y) :- r(X, Y, []),!.
r([], Z, Z).
r([H], [H|Z], Z).
r([H|[H1|T1]], W1, Z) :- r(T1, W1, [H1|[H|Z]]).
```

(by equality substitution)

Note that the straight-line predicate "reverse" is not expanded in-line because it calls the recursive predicate r.

Example 11.

THe following program optimize_1 is to delete redundant variables occurring in two consecutive equations of a clause. It reads in a clause from an input file and writes out the resulting clause to an output file. In a form of optimization schema, it realizes

$p(\mathbf{A})$:- **S**, $X = f(t_1, \ldots, t_n)$, $X = f(r_1, \ldots, r_n)$, **T**.

$p(\mathbf{A})$:- **S**, $f(t_1, \ldots, t_n) = f(r_1, \ldots, r_n)$, **T**.

where the variable "X" does not appear in $p(\mathbf{A})$, **S** and **T**.

```
optimize_1(IF, OF) :- see(IF), tell(OF), repeat, read(T),
```

```
    transform(T), seen, told.

transform('end_of_file') :- !.
transform(T) :- fold(T, R), write(R), write('.'), nl, !, fail.

fold((H:-G), (H:-R)) :- red(H, G, R).

red(H, (X=T1, Y=T2), (T1=T2)) :- var(X), X==Y,notoccur(X,[H]).
red(H, ((X=T1), ((Y=T2), T)), R) :- var(X), X==Y,
    notoccur(X, [H, T]), red(H, ((T1=T2), T), R).
red(H, (A, T), (A, Z)) :- red([H, A], T, Z).
red(H, A, A).

notoccur(X, T) :- var(T), !, not(X==T).
notoccur(X, T) :- T=..[F|As], mapnc(X, As).

mapnc(X, []).
mapnc(X, [H|T]) :- notoccur(X, H), mapnc(X, T).
```

Note that:

(1) optimize_1, transform, fold and notoccur are straight-line predicates and red and mapnc are tail-recursive predicates.

(2) transform is a-deterministic predicate.

(3) The predicate "transform(T)" occurring in "optimize_1" can not be expanded in-line since it contains a nondeterministic built-in predicate "repeat".

(4) THe predicate calls "notoccur(X, [H])" and "notoccur(X, [H, T])" occurring in the predicate definition of the predicate "red" can not be expand in-line since they does not satisfy the

conditions in Cut schema (case 2).

(5) The predicate call "notoccur(X, H)" occurrng in the predicate definition of the predicate "mapnc" can not be expanded in-inline since its defining clauses call "mapnc" again.

As the result, only the predicate call "fold(T, R)" occurring in the predicate deinition of the predicate transform is expanded in-line through the following process:

```
transform(T) :- fold(T, R), write(R), write('.'), nl, !,fail.
```

```
transform(T) :- T=(H':-G'), R=(H':-R'), red(H', G', R'),
                write(R), write('.'), nl, !, fail.
```

```
transform((H':-G')) :- red(H', G', R'), write((H':-R')),
                       write('.'), nl, !, fail.
```

6. Discussions and concluding remarks

Various optimization techniques at the source level have been presented without restricting them to pure Prolog. Some of them are logically complicated, compared with those of the optimization methods [9, 10, 11] for conventional programming languages. This seems to originate from the two-facedness of Prolog, that is, procedural and nonprocedural nature. In fact, the purely logical nature of Prolog allows to improve Prolog programs simply as logical formulas by means of obvious equivalence preserving transformation. On the other hand, the procedural interpretation of Prolog together with the specialized

execution mechanism such as left to right, top to down and backtracking control, and several non-logical language constructs such as meta or side-effect make it difficult to improve programs as logical formulas.

Our approach in improving Prolog programs is syntactical one, and is not heuristic in contrast with the program transformational approach in [3, 4]. Consequently, the method described in this paper may not lead to a qualitative and drastic improvement of a program and this seems to reveal a limitation of the syntactical approach at the source level. And yet, in the program transformation, the basic factors for qualitatively changing programs are the rules such as the introduction of definition, abstraction as introduced in [4, 3], and their applications require very heuristic knowledge or eureka.

In expanding programs in-line, we have devoted a considerable of space to show how controversial cuts should be treated. This is because a cut or its substitute would be considered to be an important language construct in improving the efficiency of problem solving programs as far as nondeterministic programming languages are concerned. Our way of treating cuts, using the determinacy, allows to extend the class of predicate definitions to which the inline expansion can be safely done.

Finally, we mention some research topics in the future which lead to the reinforcement and reliability of our present optimization method;

(1) type checking,

(2) intelligent backtracking,

(3) data/control flow analysis,

(4) formal verification of equivalence etc.

(1), (2) and (3) are mutually related and especially the type checking in untyped languages [12] may be a promising capability to extend the concepts of r-determinacy and a-determinacy introduced in Section 2. As a result, it could further promote simplifications to be done at the symbolic level of programs. The formal verification of equivalence between a program and its improved one is a very important but challenging theme. Some works on it in pure Prolog are known [3], but no works on full set of Prolog.

Acknowledgements

The authors would like to acknowledge the continuing guidance and encouragement of Dr. Tosio Kitagawa, the president of their institute.

The authors also would like to express their appreciation to their colleagues of the institute, T. Yokomori and J. Tanaka for their critical remarks and useful suggestions on earlier versions of the paper, and T. Chikayama of ICOT for discussions and helpful comments on program optimization of Prolog.

This work is part of a major R & D project of the Fifth Generation Computer, conducted under program set up by the MITI.

References

[1] D. L. Bowen : Decsystem-10 PROLOG USER'S MANUAL, version 3.43, Dept. of Artificial Intelligence, Univ. of Edinburgh 1983.

[2] Chikayama, T. : Source level optimization in logic programming languages, draft, 1983.

[3] Tamaki, H. and Sato, T. : A transformation system for logic programs which preserves equivalence, ICOT TR : TR-018, 1983.

[4] Burstall, R. M. and Darlington, J. : A transformation system for developing recursive programs, JACM, Vol. 24, No. 1, pp. 44-67, 1977.

[5] Scheifler, R. M. : An analysis of inline substitution for a structured programming language, CACM, Vol. 20, No. 9, pp. 647-654, 1977.

[6] Martelli, A. and Montanari, U. : An efficient unification algorithm, ACM TOPLAS, Vol. 4, No. 2, pp. 258-282, 1982.

[7] Aho, A. V., Sethi, R. and Ullman, J. D. : Code optimization and finite Church-Rosser systems, in Rustin, R. ed. : Design and optimization of compilers, Prentice-Hall Inc., pp. 89-105, 1972.

[8] Komorowski, H. J. : Partial evaluation as a means for inferencing data structures in an applicative language : A theory and implementation in case of PROLOG, Conf. record of the 9th ACM Symp. on Principles of programming languages, ACM, pp. 255-267, 1982.

[9] Allen, F. E. and Cocke, J : A catalogue of optimizing transformations, in R. Rustin ed. : Desigh and optimization of compilers, Prentice-Hall, pp. 1-30, 1972.

[10] Arsac, J. J. : Syntactic source to source transforms and program manipulation, CACM, Vol. 22, No. 1, pp. 43-54, 1979.

[11] Loveman, D. B. : Program improvement by source-to-source transformation, JACM, Vol. 24, No. 1, pp.121-145, 1977.

[12] Ramsay, A : Type-checking in an untyped language, Int. J. Man-Machine Studies, Vol. 20, pp. 157-167, 1984.