TR-216

UNDECIDABILITY OF DETERMINACY. TWO DECIDABLE CASES OF DETERMINACY AND THEIR APPLICATIONS TO SOURCE-TO-SOURCE TRANSFORMATION OF PROLOG PROGRAMS

by

H. Sawamura

(FUJITSU Ltd.)

November, 1986

UNDECIDABILITY OF DETERMINACY, TWO DECIDABLE CASES OF DETERMINACY AND THEIR APPLICATIONS TO SOURCE-TO-SOURCE TRANSFORMATION OF PROLOG PROGRAMS *

Hajime Sawamura

International Institute for Advanced Study of Social Information Science (IIAS-SIS), FUJITSU LIMITED, 140 Miyamoto, Numazu, Shizuoka 410-03, Japan

* This paper is a revised and extended version of a paper [1] presented at 1985 Symposium on Logic Programming, held in Boston, Massachussets, U.S.A., July 15-19, 1985.

ABSTRACT

The determinacy of a predicate call (goal) plays very important roles in source-to-source transformation (optimization) of Prolog, a nondeterministic logic programming language. By the determinacy of a predicate call, it is understood that it succeeds through at most one clause of its defining clauses when it calls them, and it never succeeds again when it is backtracked.

In this paper, we deal with three themes on determinacy. First, it is shown that the problem whether for any predicate (call) it is deterministic or not is undecidable. Its implications are then examined. Second, the concepts of a-determinacy and r-determinacy, as decidable cases of determinacy, are introduced. These concepts are mutually defined, and their properties are investigated. Third, based on these concepts, three applications to the program transformation of Prolog are described, namely, the inline expansion, the automatic cut insertion and the simplification of a sequence of conjuncts.

1. INTRODUCTION

The terms "determinacy" and "nondeterminacy" often appear in diverse branches of computer science such as automata and formal language theory, computation theory, programming languages and their semantics and verification, etc., as well as other fields of science. Although their definitions differ in the respective fields, the problems that need to decide the determinacy itself have been few except for theoretical interest. In this paper, we deal with the transformation of programs in Prolog [2] in which the determinacy of a predicate plays extremely important roles.

The nondeterministic programming languages which enable us to express the procedures with essentially nondeterministic nature have been studied by several authors. Among others, the languages devised by Floyd [3], Dijkstra [4], and micro-planner, an artificial intelligence-oriented programming language [5] are well-known nondeterministic ones, in addition to contemporary Prolog and Concurrent Prolog [6]. The programs written in these languages are nondeterministic in the two main senses : don't care and don't know [7]. Prolog and micro-planner realize "don't know" characteristic of nondeterminacy by backtracking, and Concurrent Prolog and Dijkstra's language of guarded commands realize "don't care" characteristic. The language by Floyd can have both characteristics according to interpretations of the nondeterministic construct.

This paper is concerned with nondeterminacy by means of backtracking in Prolog. By the determinacy of a predicate call (goal), it is operationally understood that it succeeds through at most one clause of its defining clauses when it calls them, and it never succeeds again when it is backtracked. With this definition, it is shown that the decision problem for such determinacy is unsolvable, and its various implications are examined. In this connection, the decision problem in the "don't care" sense of nondeterminacy would be reduced to the undecidability of the validity problem of first-order logic.

Due to the language character of Prolog, its language processing system tends to require additional time and space overhead for backtracking, compared with conventional programmig languages. One promising information for reducing it is to determine whether each predicate call is deterministically accomplished or not. It is , however, impossible in principle to determine it on the account of the unsolvability of the decision problem mentioned above. Therefore, we have to seek the concepts of algorithmically decidable determinacy. As decidable cases of determinacy, two closely related concepts, a-determinacy and r-determinacy, are introduced, and their properties are investigated. These determinacy are operationally and syntactically defined without committing to the semantics of a predicate.

Various source-to-source optimization techniques for Prolog have been presented by the author and his colleagues for the purpose of improving Prolog programs [8]. In our terminology, optimizing Prolog programs is meant to improve them in the sense of partial evaluation or symbolic execution. From the computational complexity point of view, this amounts to reducing the computation steps at the source-level to some extent. Those techniques are different from the unfold/fold transformation of programs [9, 10], the partial evaluation [11] and the deductive construction of Horn clause programs [12]. Based on those techniques a practical Prolog optimizer, which is not for pure Prolog but for full set of Prolog, has been implemented [8]. The complicated data/control flow of Prolog programs often forces us to require various preconditions in the optimization rules of programs. Of these preconditions, it is the determinacy of predicates, among others, that has been important to construct the Prolog optimizer. In fact, the determinacy of a predicate allows us to formulate the most efficient source-to-source optimization techniques. In this paper, three applications of determinacy to Prolog optimization are described : the inline expansion as an interprocedural optimization technique, the automatic cut insertion as an intraprocedural optimization technique and the deletion of multiple conjuncts in a clause as a local optimization technique.

The remainder of the paper consists of five sections. Section 2 describes the notations. Section 3 includes the proof of the undecidability of determinacy and its consequences. Section 4 provides the two decidable cases of determinacy and their properties, as well as their extensions. Section 5 includes three applications of determinacy to the source-to-source transformation of Prolog. Final section describes concluding remarks.

2. NOTATIONAL CONVENTIONS

We assume that readers are familiar with the syntax and the semantics of Prolog [2]. Here, we present only the notations and definitions needed to describe the transformation schemata, which will be introduced in the succeeding sections. It should be noted that we use the distinguished symbols as syntactical variables ranging over the syntactic domains of Prolog.

[Notational conventions]

(1) The letters P, H (with or without subscripts) represent goals (predicate calls) or heads of clauses, which are of the form of predicate names followed by some arguments, and the letters p,q,r represent predicate names or propositions.

(2) The boldface letters **S** and **T** (with or without subscripts) represent (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) The boldface letters **P** and **Q** represent vertical rows of clauses or

goals.

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

In order to illustrate that a sequence of clauses or goals is transformed into an improved one by using appropriate predicate definitions if necessary, we use a horizontal line which corresponds to derivability in logic.

[Transformation schema]

The transformation schema is a figure of the form

$$\frac{\mathbf{P}}{\mathbf{Q}}$$

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

3. UNDECIDABILITY OF DETERMINACY

Undecidable results, in general, are often derived by some suitable coding or ascribing to other undecidable results [13, 14]. Our proof, like Russell's paradox, creates a simple antinomy in terms of Prolog programs.

Before going into a proof of undecidability of the determinacy, it must be noted that computable functions are computable in Prolog, as well as in Horn clause program [15, 16].

Let us introduce the concept of determinacy to be needed in our transformation techniques. It is not a specialized one, but can be generally accepted for other purposes as well. In fact, it seems to be the same as the one described in Warren's report [17] and Mellish [18].

Definition 1. A goal (or a predicate call) is deterministic if it succeeds through at most one clause of its defining clauses when it calls them, and it never succeeds again when it is backtracked.

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.

Debray [19] consider a more general and less operational notion of predicate, that is, functionality relative to a mode, where all alternatives may produce the same result. It should be remarked that our definition of determinacy does not refer to the functionality of a predicate at all, but is only concerned with the success or failure of a predicate call by unification. In general, it is obviously undecidable if a Prolog predicate is functional. Theorem 2 below claims the undecidability of our determinacy in a more general setting of Prolog.

Theorem 2. 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. Therefore it is not deterministic.

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

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

It should be remarked that 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.

Obviously we have

Corollary 3. No algorithm exists to decide whether for any predicate call it is nondeterministic or not.

Corollary 4. 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 call as a special case, but it is impossible by Theorem 2.

Corollary 5. No algorithm exists for deciding whether for any proposition (without any variable) it is deterministic or not.

Proof. The proof for Theorem 2 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.

Theorem 6. 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. Similar to the proof of Theorem 2.

Next, we turn to the undecidability of the determinacy in "don't care" sense. In this case, we can ask a question whether or not the selection points in the possible execution paths of programs can be uniquely determined. For example, it is not decidable which guards in Dijkstra's nondeterministic language [4] are true. Obviously, such a decision problem can be reduced to the undecidability of the validity problem of first-order logic [20].

4. TWO DECIDABLE CASES OF DETERMINACY

Due to the negative result of Theorem 2, the concepts of effectively decidable deterministic goals are desired.

We introduce the two kinds of concepts of determinacy : a-determinacy (absolute determinacy) and r-determinacy (relative determinacy). Generally, the success or failure of a predicate call depends on the contents of its arguments. The r-determinacy captures such an argument dependency of a deterministic predicate call. On the other hand, a-determinacy means determinacy which does not depend on predicate arguments at all. Thus, a predicate p is a-deterministic if and only if p(**A**) is r-deterministic for every argument **A**.

In this section, first, a mutually recursive definition of these two concepts is given, and then the separate definitions of them are presented.

4.1 a-determinacy and r-determinacy

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 p be

H_1 :- $\mathbf{S}_1$.
.
.
.
H_i :- $\mathbf{S}_{i1}$, [!,] $\mathbf{S}_{i2}$., where the cut symbol "!" (if any) is the rightmost occurrence in the body.
.
.
.
H_n :- $\mathbf{S}_n$.

Then, for each i ($1 \le i \le n$), 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 program **P**, and every goal of $\mathbf{S}_{i1}$ and $\mathbf{S}_{i2}$ is a-deterministic or r-deterministic.

(2) There is no cut symbol in the body of the i-th clause, it is not the last clause in the program **P**, p(**A**) is not unifiable with any H_j ($i+1 \le j \le n$) 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.

The correctness of the definition can be stated as follows.

Corollary 7. If a goal p(**A**) is a-deteministic or r-deterministic, then it is deterministic.

Proof. By induction on the structure of the definition.

The definitions of a-determinacy and r-determinacy have been provided based 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. 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 to 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. Prescribing such a semantics for a predicate beforehand would be generally impossible if a type-system and/or a type-inference were not provided for Prolog as in [21, 22].

As mentioned in the beginning of this section, if a goal 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 goal is relatively deterministic since its predicate depends on how it is called. Hence, the predicate itself of an r-deterministic goal is sometimes called r-deterministic.

From the above observations, we have

Corollary 8. If a goal is a-deterministic, it is also r-deterministic.

Furthermore, from our definition

Corollary 9. A deterministic predicate call except built-in predicates is not determined to be a-deterministic if the number of its defining clauses is more than 2 and there exist no cut symbols in them.

Of course, as easily seen from the following program, the conditions in this corollary seem to be too strong;

```
H :- S,fail.
H :- T.
```

where no cut occurs in **S** and **T**, and every predicate call in **T** is a- or r-deterministic. Such a syntactical extension could be incorporated into our definition with no difficulty. Rather, we have preferred the definition of determinacy as general as possible.

Corollary 10. At the propositional level, that is, when the predicate to be examined is a proposition, a-determinacy coincides with r-determinacy.

Corollary 11. For every argument **A** of a predicate p, p(**A**) is r-deterministic iff a predicate p is a-deterministic.

Proof. (=>) From an arbitrariness of the argument A, consider a goal p(**X**) where **X** denotes a sequence of variables. It can not be r-deterministic with the condition (2) in the definition of r-determinacy. So, if p(**A**) is r-deterministic for every argument **A**, then it must be r-deterministic with the condition (1) or (3) in the definition of r-determinacy. Hence, it is a-deterministic. (<=) BY Corollary 8.

Lemma 12. For every argument **A** of a predicate p, a goal p(**A**) succeeds (is provable) iff a goal p(**X**) succeeds (is provable), where **X** is a sequence of variables.

Proof. Due to the following form of theorem in first-order logic :

$\vdash \forall x.p(x)$ iff $\vdash p(x)$, where x is a free variable.

Corollary 13. Let **X** be a sequence of variables, a goal p(**X**) is r-deterministic iff a predicate p is a-deterministic.

Proof. Due to Corollary 11 and Lemma 12.

Corollary 11 and Corollary 13 suggest an equivalent alternative definition of determinacy. That is, r-determinacy can be first defined and then a-determinacy in terms of Corollary 11 or Corollary 13. Formally we define r-determinacy and a-determinacy as follows.

[r-determinacy]

A predicate call p(**A**) is termed r-deterministic if it satisfies the following conditions.

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

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

H_1 :- $\mathbf{S}_1$.
.
.
.
H_i :- $\mathbf{S}_{i1}$, [!,] $\mathbf{S}_{i2}$., where the cut symbol "!" (if any) is the rightmost occurrence in the body.
.
.
.
H_n :- $\mathbf{S}_n$.

Then, for each H_i ($1 \leq i \leq n$) 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 program **P**, and every goal of $\mathbf{S}_{i1}$ and $\mathbf{S}_{i2}$ is r-deterministic.

(2) There is no cut symbol in the body of the i-th clause, it is not the last clause in the program **P**, p(**A**) is not unifiable with any H_j ($i+1 \leq j \leq n$) and every

goal of S_{i1} and S_{i2} is r-deterministic.

(3) There exist cut symbols in the body of the i-th clause and every goal of S_{i2} is r-deterministic.

[a-determinacy]

A predicate p is termed a-deterministic if for every argument A, a goal p(A) is r-deterministic, or A predicate p is termed a-deterministic if a goal p(X) is r-deterministic, where X denotes a sequence of variables.

It is noted that the former part of the definition of a-determinacy is not effective.

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

where the predicate calls p(X) and p(b) call their defining clauses above.

Example 3. The following predicate 'transform' is a-deterministic whatever the predicate 'fold' is.

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

4.2 Extensions

Here, we touch on miscellaneous methods for expanding the class of our determinacy.

(1) Simple extension : In the condition (2) of the definition of a-determinacy and r-determinacy, when a goal is unifiable with a head of the defining clauses, we propagate the unification (substitution) information to the body, in order to detect the determinacy of the goals in the body.

(2) Use of mode information : As pointed out by Mellish [18], mode information of a predicate can be expoited to expand the class of determinacy. For example, consider the following program :

```
:- mode programming_language(+,-).
computer_language(P) :-
        programming_language(P,Inventor),human(Inventor).
programming_language(lisp,mcCarthy).
programming_language(prolog,colmerauer).
programming_language(pascal,wirth).
human(mcCarthy).
human(colmerauer).
human(wirth).
```

A predicate call "programming_language(P,Inventor)" can be seen r-deterministic with the help of its mode declaration.

(3) A type-system and/or a type-inference [21, 22] would be one of promizing methods for expanding the class of our determinacy. But here, we do not go into this independent topic further since it is beyond the scope of the present paper.

5. APPLICATIONS OF DETERMINACY TO SOURCE-TO-SOURCE TRANSFORMATION OF PROLOG PROGRAMS

In this section, we describe three applications of determinacy to transformation techniques for Prolog programs.

(1) Inline expansion

In general, the main purpose of the inline expansion is twofold : to delete subroutine linkage overhead and to increase opportunities for local optimizations by providing more global program units for them.

A Prolog program has, by nature, several alternative clauses for a predicate. Due to this nondeterminacy of Prolog, the inline expansion techniques are more complicated in Prolog than ordinary programming languages. Here, we propose a natural method for the inline expansion of Prolog programs. In this method, 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 no alternative clauses have cuts in their bodies, because the cuts brought into the original clause usually cause a different control flow. The next example exhibits such a situation.

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

Thus, the existence of cut symbols in the defining clauses has a serious influence upon the possibilities of the inline expansion. In what follows, a method of the inline expansion is schematically introduced, in which the determinacy allows to expand a call by its defining clauses including cuts. In this paper, a clause with no body, say "P.", is identified with a clause "P :- true."

[Transformation schema]

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

H_1 :- S_1.
$\vdots$
H_i :- S_{i1}, $(p(A) = p(A_1)'$, T_1' ; ... ; $p(A) = p(A_m)'$, $T_m')$, S_{i2}.
$\vdots$
H_n :- S_n.
$p(A_1)$:- T_1.
$\vdots$
$p(A_m)$:- 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 program and every predicate call 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.

Taking Theorem 8 into consideration, the phrase " a-deterministic or r-deterministic " in this definition could be replaced simply by " r-deterministic ". However, the above definition is intended for such a situation that given a program (a set of predicate definitions), a-determinacy is detected beforehand as an attribute of a predicate.

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 call p(**A**) with any head p($\mathbf{A}_i$) whose clause includes cuts is known to fail. However, currently we are not concerned with these situations for simplicity.

Example 4.

```
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(Y) and p(b) call the definnig clauses given in Example 1 and 2 respectively.

The lower sequence will have to be further optimized by the use of the other transformation techniques presented in [8]. The consecutive applications of them to the second clause in the lower sequence yield

```
r(b,L,[b|L]) :- p(b),!.
```

(2) Automatic insertion of cut symbols

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

[Transformation schema]

$$\frac{\begin{array}{l} H_1 \ :- \ S_1. \\ \vdots \\ H_i \ :- \ S_{i1},\ S_{i2}. \\ \vdots \\ H_n \ :- \ S_n. \end{array}}{\begin{array}{l} H_1 \ :- \ S_1. \\ \vdots \\ Hi \ :- \ S_{i1},\ !,\ S_{i2}. \\ \vdots \\ H_n \ :- \ S_n. \end{array}}$$

where either of the following conditions is satisfied:

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

(ii) If there exist cuts 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 the same remark as the transformation schema of the inline expansion is applied to the phrase " a-deterministic or r-deterministic " in this definition.

Example 5.

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

where the predicate calls p(b) and q(X) call their defining clauses given in Example 1 and 2 respectively.

(3) Simplification of a sequence of goals

In [8], we have presented various techniques for simplifying a sequence of goals at the propositional level. Most of them are local simlification rules or deletion strategies, and are often applied to the resulting clauses after the inline expansion as well as are used individually within a clause. Here, we take the optimization schema : "deletion of multiple conjuncts in a clause" in which the determinacy of a predicate call matters.

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

[Transformation schema]

$$\frac{H :\text{-}\ S_1,\ P,\ S_2,\ P,\ S_3.}{H :\text{-}\ S_1,\ P,\ S_2,\ S_3.}$$

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

(i) P is r-deterministic.

(ii) P is not a predicate call with side effect such as a built-in input/output predicate or a meta predicate, and furthermore it is not an extra control predicate such as cut symbol, "repeat".

We show below an instance of the transformation schema which are not correct on account of the nondeterministic character of a predicate to be deleted. Suppose we have the assertions :

```
q(a).
q(b).
q(c).
```

then consider the following instance,

$$\frac{\text{repeat, q(X), repeat, not(X = a)}}{\text{repeat, q(X), not(X = a)}}$$

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.

The next instance seems to be correct although it violates the condition (i) above. But, it reveals an anomaly concerning the order of solutions. 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 consider the following instance,

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

The upper sequence terminates with X = f(b,d) and then X = f(e,V), but the lower one terminates with X = f(e,V) and then X = f(b,d). That is, both the solution sets are equal but the order of solutions is different.

6. CONCLUDING REMARKS

We have given a simple proof to the unsolvability of the determinacy. This might be proved in a constructive way such as coding, although our definition of the determinacy does not seem to be suitable to such a proof. In fact, it may be proved from the halting problem of the Turing machine [13] or Post's correspondence problem [13] , etc. and also from some results of formal language theory such as the theorems on the deterministic language or the ambiguity of a language [14].

In his book [7, Chapter 5, p.114], Kowalski writes : " The situation, however, in which search can be restricted because a procedure call computes the value of a function is undecidable in principle. It is easier for the programmer to convey such information to the program executor as a comment about the program, than it is for the executor to discover the fact for itself. " Kowalski seems to found his assertion on the undecidability of the validity problem in first-order logic [20]. His definition of determinacy is concerned with the functionality of a procedure call. For example, a relation F(x, y) is deterministic when the variable y is a function of x in the relation F(x, y) and x is given as input [7, Chapter 5, p. 113]. With this definition, the decision problem of determinacy can be obviously ascribed to the validity problem of a first-order logic formula such as

$\forall$ x, y, z [Prog -> (F(x, y) & F(x, z) -> y = z)]

where Prog is a set of Horn clauses which specifies the predicate F. However, notice that we have provided a more general definition of determinacy for Prolog program transformation than that of Kowalski, in the sense that it is only concerned with the success or failure of a predicate call, and with that definition have considered the decision problem. Theorem 2 can be thought of as giving a formal justification for his assertion in a more general setting of Prolog. The cut, on the ground of which we have put the definitions of the decidable determinacy, can be viewed as a clue of the determinacy detection given to the program executor.

Nondeterminacy is said to be one of the characterizations of nonprocedural programming [23]. Prolog, a nondeterministic logic programming language, is deeply involved in the suppression of unnecessary detail from the statement of an algorithm. For the purpose of transforming programs, however, we have shown that the detection of the determinacy permits us to improve the Prolog programs at the source-level. From the point of view of Prolog programming methodology, it may turn out to give the programmer a beneficial information on the behavior of his program, resulting in a rise of the reliability of Prolog programs. On the other hand, from the point of view of implementation issues of computer languages, it would be useful for an efficient compilation and an efficient implementation of or-parallelism as well.

Last but not least, we mention Smolka's work [24]. In relation to the determinacy, he proposes to declare a procedure as a functional procedure and to assert a procedure call as a functional call, rather than to check the determinacy. In this paper, however, we have presented a method to extract the determinacy information directly from a program text, not requiring user-supplied declarations and assertions. Conceptually, the detections of a-determinacy and r-determinacy roughly correspond to declaring a functional procedures and asserting a functional call respectively.

ACKNOWLEDGEMENTS

The author would like to express his gratitude to Dr. Tosio Kitagawa, the president of IIAS-SIS and Dr. Hajime Enomoto, the director of IIAS-SIS, for the continuing guidance and encouragement.

The author is also grateful to the former coauthor, T. Takeshima for discussions and helpful comments.

Finally, the author would like to thank R. Venken of Belgian Institute of Management for many valuable comments on his earlier research report.

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] Sawamura, H. and Takeshima, T. : Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization, Proc. of the 1985 Symposium on Logic Programming, IEEE Computer Society, Boston, Ma., pp. 200-207, 1985.

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

[3] Floyd, R. : Non-deterministic algorithms, JACM, Vol. 14, No. 4, pp. 636-644, 1967.

[4] Dijkstra, E. W. : A discipline of programming, Prentice-Hall, 1976.

[5] Sussman, G. J. : Micro-planner reference manual, MIT AI-Memo 203A, 1971.

[6] Shapiro, E. Y. : A subset of concurrent, Prolog and its interpreter, ICOT, TR-003, 1983.

[7] Kowalski, R. : Logic for problem solving, North Holland, 1979.

[8] H. Sawamura, T. Takeshima and A. Kato : Source-level optimization techniques for Prolog, IIAS Research Report No. 52, 1985.

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

[10] Tamaki, H. and Sato, T. : Unfold/fold transformation of logic programs, Proc. of the 2nd Int. Logic Programming Conf., pp. 127-138, 1984.

[11] 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.

[12] Hogger, C. J. : Derivation of logic programs, JACM, Vol. 28, No. 2, pp. 372-392, 1981.

[13] Davis, M. (editor) : The undecidable, Raven Press, 1965.

[14] Hopcroft, J. E. and J. D. Ullman : Formal languages and their relation to automata, Addison-Wesley, 1969.

[15] Tarnlund, S-A. : Horn clause computability, BIT, Vol. 17, pp. 215-226, 1977.

[16] Sebelik, J. and Stepanek, P. : Horn clause programs for recursive functions, in Clark, K. L. and Tarnlund, S-A. (editors) : Logic programming, Academic Press, pp. 325-340, 1982.

[17] Warren, D. H. D. : Implementing Prolog - compiling predicate logic programs, D.A.I. Research Reports No. 39 and No. 40, Dept. of Art. Intelligence, Univ. of Edinburgh, 1977.

[18] Mellish, C. S. : Some global optimizations for a Prolog compiler, J. of Logic Programming, Vol. 2, No. 1, pp. 43-66, 1985.

[19] Debray, S. K. and Warren, D. S. : Detection and optimization of functional computations in Prolog, LNCS 225, pp. 490-5-4, 1986.

[20] Church, A. : A note on the Entscheidungsproblem, The Journal of Symbolic Logic, Vol. 1, No. 1, pp. 40-41, 1936, and Correction to a note on the Entscheidungsproblem, ibid., Vol. 1, No. 3, pp. 101-102, 1936.

[21] Mycroft, A. and O'Keefe, R : A polymorphic type system for Prolog, Logic Programming Workshop, pp. 107-121, 1983.

[22] Mishra, P. : Towards a theory of types in Prolog, Proc. of the 1984 Int. Symp. on Logic Programming, IEEE, pp. 289-298, 1984.

[23] Leavenworth, B. M. : Nonprocedural programming, LNCS, Vol. 23, Springer, pp. 362-385, 1975.

[24] Smolka, G. : Making control and data flow in logic programs explicit, Proc. of the 1984 Lisp and Functional Programming Language Conf., ACM, pp. 311-322, 1984.