TR-088

Recursive Unsolvability of Determinacy, Solvable Cases of Determinacy and their Applications to Prolog Opetimization

Hajime Sawamura and Taku Takeshima
(Fujitsu Ltd.)

October, 1984

RECURSIVE UNSOLVABILITY OF DETERMINACY, SOLVABLE CASES OF DETERMINACY AND THEIR APPLICATIONS TO PROLOG OPTIMIZATION

Hajime SAWAMURA[*]

Taku TAKESHIMA[*]

[*] International Institute for Advanced Study of Social Information Science (IIAS-SIS), Fujitsu Ltd., Numazu, Shizuoka 410-03 JAPAN

Abstract

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

In this paper, first, it is shown that the problem whether for any predicate it is deterministic or not is recursively unsolvable. Its implications are then examined. Second, the concepts of a-determinacy and r-determinacy, as solvable cases of determinacy, are introduced. These concepts are mutually defined, and their properties are examined. Third, based on these concepts, three applications to Prolog optimization 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 optimization of programs in Prolog [1] 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 [2], Dijkstra [3], and micro-planner, an artificial intelligence-oriented programming language [4] are well-known nondeterministic ones, in addition to contemporary Prolog and Concurrent Prolog [5]. The programs written in these languages are nondeterministic in the two main senses : don't care and don't know [6]. 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 understood that at most one clause of its defining clauses succeeds when it is called, and it never succeeds again when it is backtracked. With this definition, it is shown that the decision problem for such determinacy is unsolvable. 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 recursive unsolvability of the decision problem mentioned above. Therefore, we have to seek the concepts of algorithmically decidable determinacy. As solvable cases of determinacy, two closely related concepts, a-determinacy and r-determinacy, are introduced. These are defined without committing to the semantics of a predicate.

Various source-to-source optimization techniques for Prolog have been presented by the authors for the purpose of improving Prolog programs [7]. 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 [8, 9]. Based on those techniques a practical Prolog optimizer, which is not for pure Prolog but for full set of Prolog, has been implemented [7]. The complicated data/control flow of Prolog programs often force 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 recursive unsolvability of determinacy and its consequences. Section 4 provides the two solvable cases of determinacy and their properties. Section 5 includes three applications of determinacy to the source-to-source optimization of Prolog. Final section describes concluding remarks.

2. Notational conventions

We assume that readers are familiar with the syntax and the semantics of Prolog[1]. Here, we present only the notations and definitions needed to describe the optimization scheme, 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.

(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 goals or a clause is transformed into an optimized one by using appropriate predicate definitions if necessary, we use a horizontal line which corresponds to derivability in logic.

[Optimization scheme]

The optimization scheme 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 optimization scheme respectively.

3. Recursive unsolvability of determinacy

Undecidable results, in general, are often derived by some suitable coding or ascribing to other undecidable results [10, 11]. Our proof, like Russell's paradox, creates a simple antinomy in terms of Prolog programs.

Before going into a proof of the recursive unsolvability of the determinacy, it must be noted that computable functions are computable in Horn clause programs [12,13] and hence in Prolog.

Let us introduce the concepts of determinacy to be needed in our optimization techniques. It is not a specialized one, but can be generally accepted for other purposes as well.

Definition 1. A goal (or called 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.

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:

(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 3. 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 2.

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. 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 2 arise. Therefore the predicate call r does not terminate. This implies that det#(r) does not terminate, according to the definition of the predicate r.

Next, we turn to the unsolvability 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 [3] are true. Obviously, such a decision problem can be reduced to the undecidability of the validity problem of first-order logic [14].

4. Solvable cases of determinacy

Due to the negative result of Theorem 2, the concepts of effectively decidable deterministic goals must be introduced.

We distinguish the following 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(**A**) is a built-in (evaluable) predicate of Prolog and it is deterministic, then p(**A**) is a-deterministic and r-deterministic.

(ii) Let the program **P** of the predicate p be

H_1 :- S_1.

.

.

.

H_i :- S_{i1}, [!,] S_{i2}., where the cut symbol '!' is rightmost.

.

.

.

H_n :- 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 if either (2) or else (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) If p(**A**) is unifiable with H_i, then there exist 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, otherwise vacuously true.

(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 following corollary can be easily checked.

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

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 semantics for a predicate beforehand would be obviously impossible.

Note that 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. From these 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 can not be 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 S and 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.

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 above.

5. Applications to Prolog optimization

(1) Inline expansion

In general, the main purpose of the inline expansions 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 are schematically introduced, together with the conditions which allow 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 :- S., where S is empty, consequently P :- true.

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

[Optimization scheme]

H_1 :- S_1.

.

.

.

H_i :- S_{i1}, p($\mathbf{A}$), S_{i2}.

.

.

.

H_n :- S_n.

p($\mathbf{A}_1$) :- $\mathbf{T}_1$.

.

.

.

p($\mathbf{A}_m$) :- $\mathbf{T}_m$. where cuts appear in some clause of the predicate p.

H_1 :- S_1.

.

.

.

H_i :- S_{i1}, ($p(\mathbf{A}) = p(\mathbf{A}_1)'$, T_1' ; ... ; $p(\mathbf{A}) = p(\mathbf{A}_m)'$, T_m'), S_{i2}.

.

.

.

H_n :- S_n.

$p(\mathbf{A}_1)$:- $\mathbf{T}_1$.

.

.

.

$p(\mathbf{A}_m)$:- $\mathbf{T}_m$.

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.

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

Example 3.

```
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 definig clauses given in the examples 1 and 2 respectively.

(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.

[Optimization scheme]

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 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 no cut occurs in S_{i2} and 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.

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 the example 1.

(3) Simplification of a sequence of goals

In [7], 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 scheme : "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 scheme.

[Optimization scheme]

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

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

(i) P is a-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 '.

(iii) 'not' predicate does not occur in the upper sequencé.

The following instance of the optimization scheme is not correct. Suppose we have the assertions;

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

then,

```
        repeat, q(X), repeat, not(X = a)
      ------------------------------------
        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.

We have discussed in [7] various other counterexamples of the optimization scheme, in which the above conditions are violated.

6. Concluding remarks

We have given a simple proof to the recursive 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 [10] or Post's correspondence problem [10] , etc. and also from some results of formal language theory such as the theorems on the deterministic language or the ambiguity of a language [11].

In his book [6, Chapter 5, p.114], Kowalski writes with no justification : " 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 [14]. 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 [6, 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

$$\forall\, x, y, z\ (\mathrm{Prog} \rightarrow (F(x, y)\ \&\ F(x, z) \rightarrow y = z)),$$

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 optimization 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 [15]. 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 optimizing 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 clue on the behavior of his program. 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.

Finally, we note that the type checking in untyped languages [16] may be a promising method for extending our definition of the determinacy.

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, J. Tanaka, T. Minami, A. Kato, M. Toda and H. Kano for discussions and helpful comments on earlier versions of the paper.

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] Pereira, F. ed. : CProlog user's manual, Version 1.4a, 1983.
[2] Floyd, R. : Non-deterministic algorithms, JACM, Vol. 14, No. 4, pp. 636-644, 1967.
[3] Dijkstra, E. W. : A discipline of programming, Prentice-Hall, 1976.
[4] Sussman, G. J. : Micro-planner reference manual, MIT AI-Memo 203A, 1971.
[5] Shapiro, E. Y. : A subset of concurrent Prolog and its interpreter, ICOT, TR-003, 1983.
[6] Kowalski, R. : Logic for problem solving, North Holland, 1979.
[7] H. Sawamura, T. Takeshima and A. Kato : Source-level optimization techniques for Prolog, Research Report (in preparation), IIAS-SIS, 1984.
[8] Burstall, R. M. and Darlington, J. : A transformation system for developing recursive programs, JACM, Vol. 24, No. 1, pp. 44-67, 1977.
[9] Tamaki, H. and Sato, T. : Unfold/fold transformation of logic programs, Proc. of the 2nd Int. Logic Programming Conf., pp. 127-138, 1984.
[10] Davis, M. ed. : The undecidable, Raven Press, 1965.
[11] Hopcroft, J. E. and J. D. Ullman : Formal languages and their relation to automata, Addison-Wesley, 1969.
[12] Tarnlund, S-A. : Horn clause computability, BIT, Vol. 17, pp. 215-226, 1977.
[13] Sebelik, J. and Stepanek, P. : Horn clause programs for recursive functions, in Clark, K. L. and Tarnlund, S-A. eds : Logic programming, Academic Press, 1982, pp. 325-340.
[14] 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.
[15] Leavenworth, B. M. : Nonprocedural programming, LNCS, Vol. 23, Springer, pp. 362-385, 1975.
[16] Ramsay, A : Type-checking in an untyped language, Int. J. Man-Machine Studies, Vol. 20, pp. 157-167, 1984.