ICOT Technical Report: TR-028

TR-028

Incorporating Naive Negation into PROLOG

by
Ko Sakai and Taizo Miyachi

October, 1983

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

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

INCORPORATING NAIVE NEGATION INTO PROLOG

K. Sakai and T. Miyachi

Institute for New Generation Computer Technology (ICOT)

ABSTRACT

This paper proposes an expanded version of the logic programming language Prolog, which is called Pure Prolog with Negation (PPN) and permits writing negative information. We present query response forms for PPN and their semantics and propose an execution algorithm. Furthermore, we discuss semantics and an execution algorithm for PPN as viewed from the standpoint of intuitionistic logic. These execution algorithms are based on existing Prolog systems and can be easily implemented.

1. Introduction

The resolution principle suggested by Robinson [Robinson 65] can be efficiently implemented if limited to Horn clauses, and Prolog is known as a language which applies the principle in this case [Battani73, Kowalski74, Warren77]. While Prolog programs have the advantage of being free from internal inconsistency because they are limited to definite clauses, they have a drawback in that they do not permit writing negative information. Unlike pure Prolog, actually implemented Prolog systems allow writing negation since they have special predicates such as "cut". But negation as written using cut is different from negation as we intuitively conceive it, and means "unprovability". Besides, cut is a predicate which cannot be understood without knowing how Prolog systems are controlled and explicitly introduces procedural interpretation into Prolog, thus seriously affecting its linguistic characteristics as a logic language. At the same time, it prevents Prolog from being interpreted independent of the processing system.

Since, Horn clauses per se do not preclude writing negative information, Prolog can be naturally expanded to permit writing naive (or intuitive) negation if general Horn clauses are allowed in Prolog programs. This expansion is dealt with in Section 2. The expanded language is called Pure Prolog with Negation (PPN). The incorporation of negation gives rise to a need to check the consistency of PPN since the language is not - unlike Prolog -

free from internal inconsistency, but this check is easily implementable on existing Prolog systems.

Because PPN is a programming language, it is necessary to define clearly its execution and the interpretation of the execution results. The execution of PPN, as with Prolog, can be viewed as a mechanism for giving responses to queries. Section 3 presents an algorithm for this execution, together with query response forms and their interpretations.

The interpretation of the results of PPN execution presented in Section 3 is indefinite in some cases; There are cases in which "it is unknown which of the several possible solutions given is the right solution, though it does exist among them." But if PPN is to be used as a programming language, it is desired that answers from the system should be generally definite. This desire stems from the standpoint that program execution is meaningless unless it finds a definite solution. This may be called a constructive or intuitionistic standpoint. Section 4 examines conditions for making solutions definite and presents an intuitionistic execution algorithm developed by incorporating the conditions into the algorithm described in Section 3.

2. Pure Prolog with Negation (PPN)

There are two types of Horn clauses:

(1) Clauses consisting solely of negative literals (called negative clauses)

(2) Clauses containing only one affirmative literal (called definite clauses)

Example 1.

$\neg a(X, Y)$ .....negative clause

$\neg a(b) \vee \neg c(X)$ .....negative clause

$a(X) \vee \neg b(X, Y) \vee \neg c(Y)$ .....definite clause

Programs in pure Prolog consist solely of definite clauses, but we expand pure Prolog to allow negative clauses as well. This Prolog is called Pure Prolog with Negation (PPN). A PPN program consists of two parts; the part composed solely of negative clauses is called the negative part, while the part composed solely of definite clauses is called the affirmative part.

2.1 PPN Syntax

The syntax of PPN is defined below. It assumes, however, that variables, constants, terms, and atomic formulas are known. In this paper, variables are all represented by a character string beginning with an uppercase alphabetic character, and constants (including function and predicate symbols) by a character string beginning with a lowercase alphabetic character.

<Definition 1> PPN syntax

(1) If and only if $\alpha 1, \ldots, \alpha n$ are atomic formulas,

$\alpha 1, \ldots, \alpha n$ is a negative clause.

(2) If and only if $\alpha 1, \ldots, \alpha n, \beta$ are atomic formulas,
$\beta \leftarrow \alpha 1, \ldots, \alpha n$ is a definite clause.

(3) If and only if $\gamma 1, \ldots, \gamma n$ are negative clauses,

$\gamma 1$.
.
.
.
γn .

is a negative part.

(4) If and only if $\delta 1, \ldots, \delta n$ are definite clauses,

$\delta 1$.
.
.
.
δn .

is an affirmative part.

(5) If and only if Γ is a negative part and Δ is an affirmative part,
$[\ \Gamma\ \Delta\ ]$
is a PPN program.

Adapting the above definition, Example 1 would be written;

$\leftarrow a(X,Y)$	...negative clause
$\leftarrow a(b),\ c(X)$	...negative clause
$a(X) \leftarrow b(X,Y),\ c(Y)$	...definite clause

That is, affirmative literals come to the left side of "$\leftarrow$" and negative literals (rid of the symbol $\neg$), to the right side. Intuitionally, "$\leftarrow$" may be interpreted as "if" and ",", which punctuates the right side of "$\leftarrow$", as "and", as in Prolog.

In definitions (1), (2), (3) and (4), n may be 0. In (1), however, if n=0, the negative clause $\leftarrow$ representing contradiction is obtained, which means that the program with the clause $\leftarrow$ in the negative part is inconsistent in the sense of 2.2.

2.2 PPN Consistency

As stated in the Section 1, PPN programs may involve internal inconsistency. This subsection describes the procedure for checking this inconsistency. Internal inconsistency in a PPN program means that something negated in the negative part is proved from the affirmative part. Therefore, the following is a procedure for checking the inconsistency.

<Procedure 1> PPN consistency

(1) An ordinary Prolog system is given the affirmative part (which has the same form as of ordinary Prolog program) as a program and queried about the contents of the negative part.

(2) If one of the contents negated by the clauses in the negative part are proved, it

can be said that the PPN program is inconsistent. That is, either the proved negative clause or some of the affirmative clauses used for the proof are wrong.

Example 2

← god(X), mortal(X).	god(jupiter) ←.
"God is not mortal."	"Jupiter is a god."
	god(parent(X)) ← god(X).
	"god's parent is a god."
	mortal (parent (jupiter)) ←.
	"Jupiter's parent is mortal."

If the query

? - god(X), mortal(X).

is given to an ordinary Prolog system with the affirmative part as a program, the system returns the answer

X = parent(jupiter).

because there is the following proof figure

← god(X), mortal(X)

← god(parent(X)), mortal(parent(X)) god(parent(X))←god(X)

← god (X), mortal(parent(X))

← god(jupiter), mortal(parent(jupiter)) god(jupiter)←

← mortal(parent(jupiter)) mortal(parent(jupiter))←

3. Execution of PPN as a Programming Language

Queries to pure Prolog (which may be interpreted as retrieval or program execution) correspond to negative clauses, but PPN also allows queries corresponding to definite clauses, that is, queries containing only one negative literal.

<Procedure 2> PPN execution

(1) If a query is composed solely of affirmative literals, it is treated the same as in pure Prolog. (In this case there is no need to use negative knowledge.)

(2) If a query contains a negative literal, it can be written as; ¬α, Γ, (Γ consists solely of affirmative literals) by ignoring the order of literals. Then α ← Γ is added to the affirmative part. (This process is called assertion)

(3) One of the clauses in the negative part is selected and queried about to the affirmative part, which now contains the clause asserted in (2). Each time the asserted clause, α ← Γ is used, the value substitution to each of the variables (which is called the unifier) is stored as a candidate for the answer to be returned. If execution succeeds, an answer is given in the following form;

variable 1 = value 1, ..., variable n = value n ...First unifier

∨ variable 1 = value 1', ..., variable n = value n' ...Second unifier

.

.

.

∨ variable 1 = value 1", ..., variable n = value n" ...Last unifier

If execution succeeds without using $\alpha \leftarrow \Gamma$ at all, the program is inconsistent. The meaning of the answer is that "the list includes a unifier that satisfies the query given, but existing knowledge cannot decide which one it is." (Refer to Example 4.)

(4) If execution fails or if it succeeds but another answer is requested, step (3) is repeatedly executed while backtracking. Needless to say, backtracking covers selection of clauses in the negative part.

Example 3.

Program:

←mortal(apollo).	mortal(X) ← man(X).
"Apollo is not mortal."	"man is mortal."
	man(X) ← man(parent(X)).
	"child is a man if his parent is a man."

Query: ?- ¬man(X).
"What is not man?"

Assert man(X) ← and query ?- mortal(apollo), then

$$\dfrac{\dfrac{\dfrac{\leftarrow \text{mortal(apollo)} \qquad \dfrac{\text{mortal(X)} \leftarrow \text{man(X)}}{\text{mortal(apollo)} \leftarrow \text{man(apollo)}}}{\leftarrow \text{man(apollo)}} \qquad \dfrac{\text{man(X)} \leftarrow}{\text{man(apollo)} \leftarrow}}{\leftarrow} \qquad \text{(A)}$$

is obtained and execution succeeds. The procedure stores the unifier in (A) and returns the answer

X = apollo.

If requested other answers, backtracking occurs and the answers are returned in sequence as follows:

X = parent(apollo),
X = parent(parent(apollo)),

.

.

.

Example 4.

We explain why it is necessary to present a special meaning such as that given in (3) of <Procedure 2>. Let us consider the following theorem and proof.

Theorem : There exist irrational numbers X and Y such that X^Y is a rational number.

Proof $\sqrt{2}$ is an irrational number. On the other hand, we have

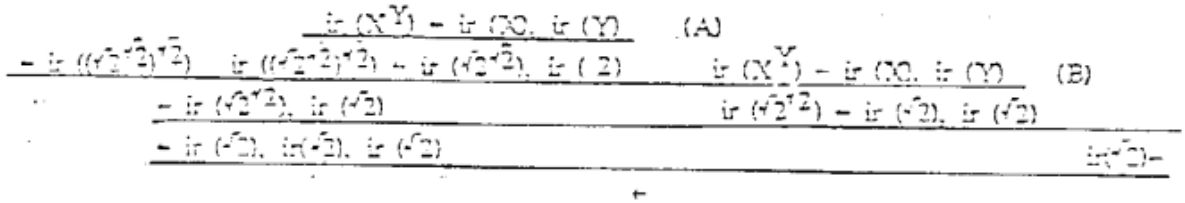
$$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = \sqrt{2}^{2} = 2$$

Therefore, $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}}$ is a rational number. Now assume that $\sqrt{2}^{\sqrt{2}}$ is an irrational number. Then, if $X = \sqrt{2}^{\sqrt{2}}$, $Y = \sqrt{2}$, X^Y is a rational number. Assume that $\sqrt{2}^{\sqrt{2}}$ is a rational number. Then, if $X = \sqrt{2}$, $Y = \sqrt{2}$, X^Y is again a rational number. Therefore, in either case there are X and Y satisfying the theorem (Q.E.D.).

The only knowledge necessary for the above proof is that $(\sqrt{2}^{\sqrt{2}})^{2}$ is a rational number and $\sqrt{2}$ an irrational number. Then let us consider the following PPN Program:

– ir $((\sqrt{2}^{\sqrt{2}})^{\sqrt{2}})$	ir $(\sqrt{2})$ –
"$(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}}$ is a rational number"	"$\sqrt{2}$ is a irrational number"

As a query, we give the above theorem, namely, ?– ¬ir (X^Y), ir (X), ir(Y). If we assert ir (X^Y) – ir (X), ir (Y) to the affirmative part and put the query ?– ir $((\sqrt{2}^{\sqrt{2}})^{\sqrt{2}})$, then we have



The above proof has two occurences of the asserted clause, whose unifier are

in (A) $X = \sqrt{2}^{\sqrt{2}}$, $Y = \sqrt{2}$ and

in (B) $X = \sqrt{2}$, $Y = \sqrt{2}$

These are just the ones corresponding to the two cases given in the earlier proof. Besides, it is evident that whether $\sqrt{2}^{\sqrt{2}}$ is an irrational number or a rational number cannot be decided from the knowledge in the program above. Therefore, which is the correct answer cannot be decided.

<Procedure 2> begins resolution with a negative clause (negative part), in line with ordinary Prolog systems, but application of the resolution principle can begin with any two clauses. Since, however, in Prolog a negative clause itself is a query in, beginning resolution with a negative clause means beginning with a query and is, therefore, efficient. In PPN, by contrast, queries may correspond to definite clauses (i.e. contain a negative literal). Beginning resolution with a negative clause in such a case is naturally expected to reduce execution efficiency. It is, therefore, important to consider an algorithm which begins resolusion with a query given. One such algorithm is presented below.

<Procedure 3> Improvement of PPN program execution efficiency

(1) For queries which do not contain negation, the procedure is the same as for Prolog.

(2) For a query containing negation ?– $a_1, \ldots, a_n$, the negative part is searched for a literal unifiable with a_1. If – B, a'_1, Δ (a' is unifiable with a_1) is found, the unifier θ which

unifies α and α' is stored as a candidate for the answer to be returned and asserts $\alpha \leftarrow \Gamma$ to the affirmative part. Subsequently, the procedure is the same as in <Procedure 2> with ?- $\theta(B, \Gamma, \Delta)$ as the goal. If the clause $\alpha \leftarrow \Gamma$ asserted is used later, the unifier is added to the list of candidates for the answer.

(3) If (2) fails (that is, no literal unifiable with α exists in the negative part), the affirmative part is searched for a literal in a body unifiable with α. If $\beta \leftarrow B, \alpha', \Delta$ is found, the unifier stored as a candidate for the answer and the process (4) is executed with ?- $\theta(\neg\beta, B, \Gamma, \Delta)$ as the goal.

(4) The negative part is searched for a literal unifiable with $\theta(\beta)$. If one is found, the procedure is the same as in (2), except that the unifier related to $\theta(\beta)$ is not added to the list of candidates for the answer.

(5) If (4) fails, the affirmative part is searched for a literal in a body unifiable with $\theta(\beta)$. The subsequent procedure is the same as in (3), except that the unifier related to $\theta(\beta)$ is not added to the list of candidates for the answer.

(6) If the goal ends in success in the above process, the program outputs the candidates for the answer as in <Procedure 2>. If the goal ends in failure or another answer is requested, the procedure backtracks.

4. PPN and Intuitionistic Logic

The proof in Example 4 uses the law of excluded middle and is often cited as an example of the unprovable in intuitionistic logic. So the PPN system seems beyond intuitionistic logic. The relation between PPN and intuitionistic logic is discussed here.

The resolution principle as a theorem prover was originally an inference rule in classical logic and has no direct relation with intuitionistic logic. In fact, application of the resolution principle requires transforming logical formulas into Skolem standard form, but intuitionistic logic does not allow such transformation.
In Prolog or in PPN , however, Horn clauses, can also be interpreted in intuitionistic logic as follows.

(1) Negative clause $\leftarrow \alpha 1, \ldots, \alpha n$ means

$$\forall X1 \ldots \forall Xm \ \neg(\alpha 1 \wedge \ldots \wedge \alpha n)$$

(2) Definite clause $\beta \leftarrow \alpha 1, \ldots, \alpha n$ means

$$\forall X1 \ldots \forall Xm \ (\alpha 1 \supset \ldots \supset \alpha n \supset \beta)$$

($X1, \ldots, Xm$ represents all variables appearing in $\alpha 1, \ldots, \alpha n, \beta$.)

Under this interpretation, the resolution principle is a valid inference rule even in intuitionistic logic (though unlike in classical logic, not complete). Especially, since Prolog programs do not have (1), changing the interpretation of (2) to

(2') $\forall X1 \ldots \forall Xm \ (\alpha 1 \supset \ldots \supset \alpha n \supset \beta)$

makes the resolution principle consistent with positive implicationistic logic, a subsystem of intuitionistic logic. ((2) and (2') are equivalent formulas in the intuitionistic logic.)

The proof in Example 4, however, on the face of it, seems to signify that PPN cannot be kept within the intuitionistic logic, but actually that is not the case. What PPN proves in Example 4 is that the three expressions

$\neg ir((\neg 2^{\neg 2})^{\neg 2})$, $ir(\neg 2)$, $\forall X \forall Y(ir(X) \wedge ir(X \supset ir(X^Y))$

result in contradiction. From this it follows that

(3) $\neg \forall X \forall Y(ir(X) \wedge ir(X) \supset ir(X^Y))$

is deducible from $ir((\neg 2^{\neg 2})^{\neg 2})$ and $ir(\neg 2)$. And in classical logic (3) is allowed to be interpreted as

(3') $\exists X \exists Y(\neg ir(X^Y) \wedge ir(X) \wedge ir(Y))$.

That is, PPN remains within the scope of intuitionistic logic if we regard what Example 4 proves as (3), not (3'), and goes beyond the scope of intuitionistic logic if we regard it as(3'). Therfore, the interpretation in <Procedure 2> is not intuitionistic.

One reason for Prolog's success as a programming language is the fact that the existential interpretation like (3') is possible even under intuitionism by confining it in the framework of definite clauses as knowledge and negative clauses as query, thus enabling it to be viewed as mechanism for finding definite solutions. But the existential interpretation under intuitionism is possible also for PPN, if the clause asserted is used only once in the proof, as in Example 3. In such a case, indefinite interpretation, as in Examples 4, become unnecessary. The following is a generalization of this.

< Theorem 1 >

Let H be a set of Horn clauses, and assume that contradiction is derived from $\alpha \leftarrow \Gamma$ and H by the resolution principle. If all the occurences of substitution to the clause in $\alpha \leftarrow \Gamma$ appearing in the proof are unifiable, then

$\exists X \ldots \exists Xm\ (\neg\alpha \wedge \Gamma)$ (X, ..., Xm represent all variables appearing in α, Γ.)

is provable from H in intuitionistic logic.

< Proof >

Let θ be a unifier that unifies all occurences of substitution to $\alpha \leftarrow \Gamma$. Let H' be the results of all necessary substitutions made to H in the proof. Then, from $\theta(\alpha \leftarrow \Gamma)$ and $\theta(H')$, contradiction can be derived by the resolution principle without substitution. From this it follows (the proof is shown later as <Lemma 1>) that

(1) $\theta(\Gamma)$ is derived from $\theta(H')$ by the resolution principle and
(2) Contradiction is derived from $\theta(H')$ and $\theta(\alpha)$ by the resolution principle.

Since the resolution principle is also valid in intuitionistic logic, it follows from (2) that $\neg\theta(\alpha)$ is derived from $\theta(H')$ in intuitionistic logic. Therefore, from $\theta(H')$ are derived $\theta(\neg\alpha \wedge \Gamma)$ and, furthermore,

$\exists X1 \ldots \exists Xm\ (\neg\alpha \wedge \Gamma)$.

And obviously, $\sigma(\Gamma)$ is derived from H. Therefore, $\exists X1 \ldots \exists Xm(\neg\alpha \wedge \Gamma)$ is derived from H. Q.E.D.

From the above theorem, the following algorithm for PPN execution in intuitionistic logic is obtained by modifying <Procedure 2> or <Procedure 3>.

<Procedure 4> Intuitionistic PPN execution

(1) The process leading to discovery of an answer is the same as in <Procedure 2 > and <Procedure 3 >.
(2) If candidates for the answer are obtained, the program checks whether they are unifiable.
(3) If they are unifiable, the program returns their most general unifier as the answer. If they are not unifiable, the program backtracks.

Example 5.

In Example 3, man(X) ← asserted is used only once to derive each answer, as stated earlier. Therefore, the answers in Example 3 may also be regarded as answers in intuitionistic execution.

Example 6.

$X = \sqrt{2}^{\sqrt{2}}$, $Y = \sqrt{2}$ and $X = \sqrt{2}$, $Y = \sqrt{2}$ in Example 4 are not regarded as answers because $\sqrt{2}^{\sqrt{2}}$ and $\sqrt{2}$ substituted to X cannot be unified.

Example 7.

← rus∈X, X∈X
"Members of the set rus do not include themselves."

To put the query ?- ¬Y∈Z to this program, we assert Y∈Z ← and query ?- r∈X.

```
                    Y∈Z ←      (A)
← X∈rus, X∈X      X∈rus ←          Y∈Z ←        (B)
        ← X∈X                     X∈X ←
                  ←
```

The substitution in (A) is Y = X, Z = rus, and in (B), Y = X, Z = X. Since, however, these can be unified, X = Y = Z = rus. Therefore, Y = rus, Z = rus is returned.

We demonstrate the truth of (1) and (2) in the proof of <Theorem 1>.

< Lemma 1 >

Let H be a set of Horn clauses. If contradiction can be derived from $\alpha \leftarrow \Gamma$ and H by the resolution principle without substitution, then

(1) Γ is derived from H and by the resolution principle.

(2) Contradiction is derived from $\alpha \leftarrow \Gamma$ and H.

< Proof >

Since (2) is obvious, we prove (1). There are only two types of resolution from Horn clauses:

A) Resolution from two definite clauses (d-resolution)

$\alpha \leftarrow B, \Gamma, \Delta$ is derived from $\alpha \leftarrow B, \beta, \Delta$ and $\beta \leftarrow \Gamma$.

B) Resolution from a negative clause and a definite clause (n-resolution)

$\leftarrow B, \Gamma, \Delta$ is derived from $\leftarrow B, \beta, \Delta$ and $\beta \leftarrow \Gamma$.

A proof of contradiction from Horn clauses using the resolution principle can be rewritten into a proof of contradiction by n-resolution only. The procedure for this is as follows. Contradiction, which is a special negative clause, cannot be proved by d-resolution alone. Therefore, if a d-resolution exists in a proof, there exist one which connects with an n-resolution, as illustrated below.

$$\dfrac{\dfrac{(A)}{\leftarrow A, \alpha, E} \quad \dfrac{\dfrac{(B)}{\alpha \leftarrow B, \beta, \Delta} \quad \dfrac{(C)}{\beta \leftarrow \Gamma}}{\alpha \leftarrow B, \Gamma, \Delta}\ \text{d-resolution}}{\dfrac{\leftarrow A, B, \Gamma, \Delta, E}{(D)}}\ \text{n-resolution}$$

If that part is rewritten into two n-resolutions as follows, the number of d-resolutions is reduced.

$$\dfrac{\dfrac{(A)}{\leftarrow A, \alpha, E} \quad \dfrac{(B)}{\alpha \leftarrow B, \beta, \Delta}\ \text{n-resolution}}{\dfrac{\leftarrow A, B, \beta, \Delta, E \quad \beta \leftarrow \Gamma}{\leftarrow A, B, \Gamma, \Delta, E}\ \text{n-resolution}}$$
$$(D)$$

This procedure is repeated until the number of d-resolutions is reduced to zero.

The proof of contradiction from H and $\alpha \leftarrow \Gamma$ is rewritten into a proof of contradiction from n-resolution alone. Then the proof takes the form that definite clauses of H or $\alpha \leftarrow \Gamma$ are applied sequentially to negative clauses. Let the part where $\alpha \leftarrow \Gamma$ is applied last be

$$\dfrac{\dfrac{(B)}{\leftarrow B, \alpha, \Delta} \quad \alpha \leftarrow \Gamma}{\dfrac{\leftarrow B, \Gamma, \Delta}{(A)}}$$

The part (A) represents a proof of contradiction from $\leftarrow B, \Gamma, \Delta$ and definite clauses.

of H alone and has no substitution. It follows that ← B, Γ, Δ is derived from definite clauses of H alone by the following <Lemma 2>. Therefore, (1) holds. Q.E.D.

< Lemma 2 >

If contradiction is proved from ← Γ and a set P of definite clauses by the resolution principle without substitution, then Γ is proved from P.

< Proof >

We use mathematical induction related to the number of steps in the proof of contradiction.

1) If the number of steps is zero, ← Γ is nothing other than ←. In this case, Γ is empty and there is nothing to prove from P. Therefore, the lemma holds.
2) Let us assume that the number of steps = n > 0 and the lemma holds for those proofs whose number of steps is less than n. In this case Γ is not empty, and the part where the resolution principle is first applied to ← Γ is of the form

$$\dfrac{\dfrac{\leftarrow \Gamma 1,\ \alpha,\ \Gamma 2 \qquad \dfrac{(B)}{\alpha \leftarrow \Delta}}{\leftarrow \Gamma 1,\ \Delta,\ \Gamma 2}}{(A)} \qquad (\ \Gamma = \Gamma 1,\ \alpha,\ \Gamma 2)$$

Here the part (A) of the proof has fewer steps than n and represents a proof of contradiction from ← Γ1, α, Γ2 and P. Therefore, by the hypothesis of induction, ← Γ1, α, Γ2 is proved from P. On the other hand, the part (B) shows a proof of α ← Δ from P. Since Δ is known to be provable from P, α is also provable from P. Therefore, Γ1, α, Γ2, namely, Γ is proved form P. Q.E.D.

Acknowledgements

The authors wish to express their thanks to K. Fuchi, director, T. Yokoi, and K. Furukawa, laboratory chiefs, of the Institute for New Generation Computer Technology (ICOT) for providing the opportunity to conduct this research. Thanks are also due to S. Kunifuji, K. Mukai, T. Kurokawa and M. Asoo of ICOT for their helpful suggestions.

[References]

[Kowalski74] Kowalski, R.: "Predicate logic is a programming language," ifip 74, North-Holland, pp. 569–574, 1974.

[Battani73] Battani G. and Meloni, H.: "Interpreteur du langage de programmation PROLOG," Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Universite d'Aix-Marseille, 1973.

[Warren77] Warren, D.H.D.: "Implementing PROLOG - compiling predicate logic programs," Research Report 39 and 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

[Robinson65] Robinson, J.A.: "A machine oriented logic based on the resolution principle," J.ACM 12, No. 1, pp. 23–41, 1965.

APPENDIX: PROLOG PROGRAM FOR PPN EXECUTION

A1. PPN inconsistency check

A1.1. program

```
:- public solve_c/1.          % inconsistency check
:- mode solve_c(+).
solve_c(DB) :- !,
        solve_c_negative_clause_select(DB,DB,A),
        nl, write('  '), write_or(A), nl.

:- mode solve_c_negative_clause_select(+,+,-).
solve_c_negative_clause_select([(:-BR)|L],DB,[(:-Bb)|A]):-
        copy_term(BR,Bb),
        solve_c_body(Bb,DB,[],A).
solve_c_negative_clause_select([_|L],DB,A):-
        solve_c_negative_clause_select(L,DB,A).

:- mode solve_c_body(?,+,?,?).
solve_c_body([],_,A,A).
solve_c_body([B|BR],DB,AA,A):-
        solve_c_literal(B,DB,DB,C,A),
        solve_c_body(BR,DB,AA,C).

:- mode solve_c_literal(?,+,+,?,?).
solve_c_literal(B,[(HR:-BR)|L],DB,AA,[(B:-Bb)|A]):-
        copy_term((HR:-BR),(Hh:-Bb)), unify(B,Hh),
        solve_c_body(Bb,DB,AA,A).
solve_c_literal(B,[_|L],DB,AA,A):-
        solve_c_literal(B,L,DB,AA,A).
```

A1.2. execution example

```
|    solve_c([(:-[god(X),mortal(X)]),
|             (god(jupiter):-[]),
|             (god(parent(X)):-[god(X)]),
|             (mortal(parent(jupiter)):-[])  ]).

| (:-[god(parent(jupiter)),mortal(parent(jupiter))])
| (god(parent(jupiter)):-[god(jupiter)])
| (god(jupiter):-[])
| (mortal(parent(jupiter)):-[])
```

```
|    :-op(500,xfx,on).
| ?- solve_c(
|        [(:-[blue(X),green(X)]),
|         (b on a :-[]),
|         (c on b :-[]),
|         (green(c) :- []),
|         (blue(a) :- [])   ]).

no
```

```
| ?- solve_c(
|          [ (:-[ir((root2**root2)**root2)]),
|            (ir(root2):-[])]).

no
```

A2. Classical execution of PPN

A2.1. program

```
:- op(900,fx,~).                % the negation symbol

:- public solve/2.              % classical PPN execution
:- mode solve(+,+).             % solve(Query, PPN_program)
solve([~H|B],DB) :- !, copy_term((H:-B),(Hh:-Bb)),
        solve_head_clause_select(Hh,DB,DB,(H:-B),[],AA),
        solve_body_asserted_db(Bb,DB,(H:-B),AA,A),
        nl, write('  '), write([~Hh|Bb]), write_or(A).
solve(B,DB):- copy_term(B,Bb),solve_body_original_db(Bb,DB),
        nl, write('  '), write(Bb), nl.

:- mode solve_head_clause_select(?,+,+,+,+,-).
solve_head_clause_select(H,[(HR:-BR)|L],DB,C,AA,A):-
        copy_term((HR:-BR),(Hh:-Bb)),
        solve_head_affirmative(H,Hh,Bb,DB,C,AA,A).
solve_head_clause_select(H,[(:-BR)|L],DB,C,A,B):-
        copy_term(BR,Bb),
        solve_head_negative(H,Bb,DB,C,A,B).
solve_head_clause_select(H,[_|L],DB,C,A,B):-
      . solve_head_clause_select(H,L,DB,C,A,B).

:- mode solve_head_negative(?,?,+,+,+,-).
solve_head_negative(H,[Hh|BR],DB,C,AA,A):- unify(H,Hh),
        solve_body_asserted_db(BR,DB,C,AA,A).
solve_head_negative(H,[B|BR],DB,C,AA,A):-
        solve_head_negative(H,BR,DB,C,AA,A),
        solve_literal_original_db(B,DB,DB).

:- mode solve_head_affirmative(?,?,?,+,+,+,-).
solve_head_affirmative(H,HR,[Hh|BR],DB,C,AA,A):- unify(H,Hh),
        solve_head_clause_select(HR,DB,DB,C,AA,C),
        solve_body_asserted_db(BR,DB,C,C,A).
solve_head_affirmative(H,HR,[B|BR],DB,C,AA,A):-
        solve_head_affirmative(H,HR,BR,DB,C,AA,A),
        solve_literal_original_db(B,DB,DB).
```

```
:- mode solve_body_asserted_db(?,+,+,+,-).
solve_body_asserted_db([],_,_,A,A).
solve_body_asserted_db([B|BR],DB,C,AA,A):-
        solve_literal_asserted_db(B,DB,DB,C,AA,C),
        solve_body_asserted_db(BR,DB,C,C,A).

:- mode solve_literal_asserted_db(?,+,+,+,+,-).
solve_literal_asserted_db(B,[(HR:-BR)|L],DB,C,AA,A):-
        copy_term((HR:-BR),(Hh:-Bb)), unify(B,Hh),
        solve_body_asserted_db(Bb,DB,C,AA,A).
solve_literal_asserted_db(B,[_|L],DB,C,AA,A):-
        solve_literal_asserted_db(B,L,DB,C,AA,A).
solve_literal_asserted_db(B,[],DB,(H:-BB),AA,[[~B|BBb]|A]):-
        copy_term((H:-BB),(Hh:-BBb)), unify(B,Hh),
        solve_body_asserted_db(BBb,DB,(H:-BB),AA,A).

:- mode solve_literal_original_db(?,+,+).
solve_literal_original_db(B,[(HR:-BR)|L],DB) :-
        copy_term((HR:-BR),(Hh:-Bb)), unify(B,Hh),
        solve_body_original_db(Bb,DB).
solve_literal_original_db(B,[_|L],DB):-
        solve_literal_original_db(B,L,DB).

:- mode solve_body_original_db(?,+).
solve_body_original_db([],_).
solve_body_original_db([B|BR],DB):-
        solve_literal_original_db(B,DB,DB),
        solve_body_original_db(BR,DB).

:- mode write_or(+).
write_or([]):- nl.
write_or([Candidate|A]):- nl, write('| '), write(Candidate), write_or(A).
```

A2.2. execution example

```
| ?- :-op(500,xfx,in).
| ?- solve([~(X in Y)],[(:-[X in X, X in russel])]).

  [~_434 in _434]
| [~_434 in russel]
```

```
| ?-  solve([~man(X)],
|           [(:-[mortal(apollo)]),
|            (mortal(X):-[man(X)]),
|            (man(X):-[man(parent(X))])]).

  [~man(apollo)]

  [~man(parent(apollo))]

  [~man(parent(parent(apollo)))]

  [~man(parent(parent(parent(apollo))))]
```

```
| ?- :-op(100,xfy,**).
| ?- solve([~ir(X**Y),ir(X),ir(Y)],
|          [ (:-[ir((root2**root2)**root2)]),
|            (ir(root2):-[])]).

  [~ir((root2**root2)**root2),ir(root2**root2),ir(root2)]
| [~ir(root2**root2),ir(root2),ir(root2)]
```

```
| ?- :-op(500,xfx,on).
| ?- solve([~green(X),green(Y),Y on X],
|        [(:-[blue(X),green(X)]),
|         (b on a :-[]),
|         (c on b :-[]),
|         (green(c) :- []),
|         (blue(a) :- [])   ]).

  [~green(a),green(b),b on a]
| [~green(b),green(c),c on b]
```

A3. Intuitionistic execution of PPN

A3.1. program

```
:- public solve_i/2.          % intuitionistic PPN execution
:- mode solve_i(+,+).         % solve_i(Query, PPN_program)
solve_i([~H|B],DB):- copy_term((H:-B),(Hh:-Bb)),
        solve_i_head_clause_select(Hh,DB,DB,(Hh:-Bb)),
        solve_i_body_asserted_db(Bb,DB,(Hh:-Db)),
        nl, write('  '), write([~Hh|Bb]), nl.
solve_i(B,DB):- copy_term(B,Bb),solve_i_body_original_db(Bb,DB),
        nl, write('  '), write(Bb), nl.

:- mode solve_i_head_clause_select(?,+,+,?).
solve_i_head_clause_select(H,[(HR:-BR)|L],DB,Q):-
        copy_term((HR:-BR),(Hh:-Bb)),
        solve_i_head_affirmative(H,Hh,Bb,DB,Q).
solve_i_head_clause_select(H,[(:-BR)|L],DB,Q):-
        copy_term(BR,Bb),
        solve_i_head_negative(H,Bb,DB,Q).
solve_i_head_clause_select(H,[_|L],DB,Q):-
        solve_i_head_clause_select(H,L,DB,Q).

:- mode solve_i_head_negative(?,?,+,?).
solve_i_head_negative(H,[Hh|BR],DB,Q):- unify(H,Hh),
        solve_i_body_asserted_db(BR,DB,Q).
solve_i_head_negative(H,[B|BR],DB,Q):-
        solve_i_head_negative(H,BR,DB,Q),
        solve_i_literal_original_db(B,DB,DB).

:- mode solve_i_head_affirmative(?,?,?,+,?).
solve_i_head_affirmative(H,HR,[Hh|BR],DB,Q):- unify(H,Hh),
        solve_i_head_clause_select(HR,DB,DB,Q),
        solve_i_body_asserted_db(BR,DB,Q).
solve_i_head_affirmative(H,HR,[B|BR],DB,Q):-
        solve_i_head_affirmative(H,HR,BR,DB,Q),
        solve_i_literal_original_db(B,DB,DB).

:- mode solve_i_body_asserted_db(?,+,?).
solve_i_body_asserted_db([],_,_).
solve_i_body_asserted_db([B|BR],DB,Q):-
        solve_i_literal_asserted_db(B,DB,DB,Q),
        solve_i_body_asserted_db(BR,DB,Q).

:- mode solve_i_literal_asserted_db(?,+,+,?).
solve_i_literal_asserted_db(H,[(HR:-BR)|L],DB,Q):-
        copy_term((HR:-BR),(Hh:-Bb)), unify(H,Hh),
        solve_i_body_asserted_db(Bb,DB,Q).
solve_i_literal_asserted_db(H,[_|L],DB,Q):-
        solve_i_literal_asserted_db(H,L,DB,Q).
solve_i_literal_asserted_db(H,[_.DB.(Hh:-B)]):- unify(H,Hh),
        solve_i_body_asserted_db(B,DB,(H:-B)).
```

```
:- mode solve_i_literal_original_db(?,+,+).
solve_i_literal_original_db(B,[(HR:-BR)|L],DB) :-
        copy_term((HR:-BR),(Hh:-Bb)), unify(B,Hh),
        solve_i_body_original_db(Bb,DB).
solve_i_literal_original_db(B,[_|L],DB):-
        solve_i_literal_original_db(B,L,DB).

:- mode solve_i_body_original_db(?,+).
solve_i_body_original_db([],_).
solve_i_body_original_db([B|BR],DB):-
        solve_i_literal_original_db(B,DB,DB),
        solve_i_body_original_db(BR,DB).
```

A3.2. execution example

```
|    :-op(500,xfx,in).
| ?- solve_i([~(X in Y)],[(:-[X in X, X in russel])]).

  [~(russel in russel)]
```

```
| ?- :-op(100,xfy,**).
| ?- solve_i([~ir(X**Y),ir(X),ir(Y)],
|          [ (:-[ir((root2**root2)**root2)]),
|            (ir(root2):-[])]).

no
```

```
| ?-  solve([~man(X)],
|          [(:-[mortal(apollo)]),
|           (mortal(X):-[man(X)]),
|           (man(X):-[man(parent(X))])]).

  [~man(apollo)]

  [~man(parent(apollo))]

  [~man(parent(parent(apollo)))]

  [~man(parent(parent(parent(apollo))))]

  [~man(parent(parent(parent(parent(apollo)))))]
```

A4. Utility progarms

A4.1. create new term with renamed variable

```
:- public copy_term/2.
:- mode copy_term(+, ?).
copy_term(Old_instance, New_instance) :-
        copy_term(Old_instance, New_instance, [], Varlist).

:- mode copy_term(+, ?, +, -).
copy_term(Old, New, Cur_Varlist, New_Varlist) :-
        var(Old), !, copy_var(Old, New, Cur_Varlist, New_Varlist).
copy_term(Old, New, Cur_Varlist, New_Varlist) :- !,
        Old =.. [F|Old_list],
        copy_list(Old_list, New_list, Cur_Varlist, New_Varlist),
        New =.. [F|New_list].

:- mode copy_list(+, ?, +, -).
copy_list([], [], Var_list, Var_list) :- !.
copy_list([Old|Old_list], [New|New_list], Cur_varlist, New_varlist) :- !,
        copy_term(Old, New, Cur_varlist, Varlist),
        copy_list(Old_list, New_list, Varlist, New_varlist).

:- mode copy_var(+, ?, +, -).
copy_var(Old, New, [], [(Old,New)]) :- !.
copy_var(Old, New, [(X, New)|Varlist], [(X, New)|Varlist]) :-  Old == X, !.
copy_var(Old, New, [Pair|Cur_Varlist], [Pair|New_Varlist]) :- !,
        copy_var(Old, New, Cur_Varlist, New_Varlist).
```

A4.2. unification with occur check

```
:- public unify_l/2.
:- mode unify_l(?,?).
unify_l([],[]) :- !.
unify_l([X|L],[Y|M]) :- !, unify(X,Y), unify_l(L,M).

:- public unify/2.
:- mode unify(?,?).
unify(X,Y) :- X == Y, !.
unify(X,Y) :- var(X), !, occur_check(X,Y), X = Y.
unify(X,Y) :- var(Y), !, occur_check(Y,X), X = Y.
unify(X,Y) :- !, X =.. [F|XA], Y =.. [F|YA], unify_l(XA,YA).

:- mode occur_check(+,+).
occur_check(X,Y) :- X == Y, !, fail.
occur_check(X,Y) :- var(Y), !.
occur_check(X,Y) :- !, Y =.. [F|A], occur_check_l(X,A).

:- mode occur_check(+,+).
occur_check_l(X,[]) :- !.
occur_check_l(X,[Y|A]) :- !, occur_check(X,Y), occur_check_l(X,A).
```