Kragujevac J. Math. 23 (2001) 119–130.

ONE IMPLEMENTATION OF PL PROVER ALGORITHM

Tatjana Timotijević

Faculty of Science, P. O. Box 60, 34000 Kragujevac, Yugoslavia

(Received July 2, 2001)

Abstract. In this paper a practical solution to the problem of implementing PL prover algorithm is offered in a form of a program written in C-language.

1. INTRODUCTION

J. Robinson (1965) found the resolution rule, based on which many of theorem provers were developed. One of basic faults of resolution provers consists in almost impossible implementation of some strategy to enable relevance of steps. Such a strategy would save us from repeating a large number of new cases, which is the major flaw of these provers.

On the other hand, Prolog algorithm in its base contains relevance, and thus, it is very successful in wide area of problems. The problem with Prolog is that it is not applicable to any set of disjunctions, but only for those known as Horn formulas.

Considering all these problems, logic programming has evolved for 30 years with an idea of making some algorithm, similar to PROLOG, but applicable to all disjunctions. In 1997 S. Prešić has developed PL proof procedure [2], that accomplished both

relevance of PROLOG, and the ability to be applied to any set of disjunctions. In the case of Horn formulas, this algorithm is almost identical to PROLOG algorithm.

In Section 2 of this paper we talk about the propositional PL proof procedure. Section 3 brings a few examples and explains expansion to the first order case. The implementation of the alogorithm and explanation of program are subjects of the Section 4. In Section 5 we give some testing results. Section 6 brings the Conclusion.

2. ABOUT PL PROOF PROCEDURE

We give a short overview based on [2].

Lemma 1. *Let F be any set of propositional formulas not containing the atom p, and let $\phi_1(p)$, $\phi_2(p)$, ... be propositional formulas containing p. Then we have the following equivalences:*

$$\begin{array}{rcl} F, \phi_1(p), \phi_2(p), \ldots \vdash p & \leftrightarrow & F, \phi_1(\bot), \phi_2(\bot), \ldots \vdash \bot \\ F, \phi_1(p), \phi_2(p), \ldots \vdash \neg p & \leftrightarrow & F, \phi_1(\top), \phi_2(\top), \ldots \vdash \bot \ . \end{array} \tag{1}$$

Lemma 2. *The equivalence*

$$F, p_1 \vee \ldots \vee p_k \vdash \bot \leftrightarrow F \vdash \neg p_1, \ldots F \vdash \neg p_k (p_i \textit{ is any literal}) \tag{2}$$

is true.

The following set of rules easily follows from Lemma 1 and Lemma 2:

(R1) $F, \bot \vdash \bot \leftrightarrow \vdash \top$

(R2) $F, \phi_1(p), \phi_2(p), \ldots \vdash p \leftrightarrow F, \phi_1(\bot), \phi_2(\bot), \ldots \vdash \bot$

$F, \phi_1(p), \phi_2(p), \ldots \vdash \neg p \leftrightarrow F, \phi_1(\top), \phi_2(\top), \ldots \vdash \bot$

(R3) $F, p_1 \vee \ldots \vee p_k \vdash \bot \leftrightarrow F \vdash \neg p_1, \ldots F \vdash \neg p_k$

(R4) $F, \top \vdash A \leftrightarrow F \vdash A$ (A is a literal or the symbol $\bot$)

The main theorem about PL system is the following:

Theorem 1. *Let F be a set of clauses and ψ a literal or the symbol $\bot$. ψ is a logical consequence of the set F if and only if starting with $F \vdash \psi$ and applying the rules (R1)-(R4) a finite number of times one can obtain the sequent $\vdash \top$.*

3. A FEW EXAMPLES AND A FIRST ORDER CASE EXPANSION

Having in mind Theorem 1 one can describe the PL proof procedure for sequents of the form $F \vdash \phi$. It begins by applying the inference rule (R2) and then has the following setps:

(3) *After applying rule (R2) we get the following sequent* $F, f_1, f_2, \ldots \vdash \bot$. *Rule (R3) will be first applied to the clause* f_1, *called relevant clause, after that, if needed, to the relevant clause* f_2, *and so on.*

(4) *First, we should find the place at which the previous relevant clause was activated. Second, this clause should be omitted, and third, from this point we should continue the procedure by employing the new relevant clause.*

(5) *The sequent* $\Phi \vdash \psi$ *is replaceable if* ψ *does not occur in* Φ *and also at least one clause of* Φ *is relevant clause.*

(6) *In the procedure PL we apply* (4) *just in case when in certain step we have a replaceable sequent.*

Step (4) PL proof procedure enables using other relevant clauses in the case that using a previous one gives no satisfactory result. If while using described procedure at some step we can apply rule (R1) the procedure stops with the conclusion that ϕ is logical consequence of F. Also, if we obtain the sequent $\vdash \bot$ the procedure stops with the conclusion that ϕ is not logical consequence of F. Procedure stops with the same conclusion if we obtain seqent $\Phi \vdash \phi$ where Φ contains no literal ϕ (but it may contain its negation $\neg\phi$) and no relevant clauses.

Let us explain procedure PL in the following example.

Example 1. Prove the sequent:

$(*)$ $p \vee \neg a \vee \neg b, a \vee \neg c, a \vee \neg d, d, p \vee \neg r \vee \neg d, r \vee \neg d \vdash p$

Proof.

$(*) \leftrightarrow [1]\neg a \vee \neg b, [1]\neg r \vee \neg d, a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash \bot$

(We apply rule (R2) on the given formula. That gives new relevant clauses. These clauses are being put in front of the sequent, and in front of

them we put "address" to label the place at which these clauses appeared. We call this place *Origin-place* of the relevant clause.)

$\leftrightarrow [1]\neg r \vee \neg d, a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]a$ and

$[1]\neg r \vee \neg d, a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]b$

(The rule (R3) is applied, and since a and b originate from $[1]\neg a \vee \neg b$, we put the address [1] in front of both of them)

$\leftrightarrow [2]\neg c, [2]\neg d, [1]\neg r \vee \neg d, d, r \vee \neg d \vdash \bot$

((R2) produces two new relevant clauses which we put in front, and for them we use new address [2])

and $[1]\neg r \vee \neg d, a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]b$

$\leftrightarrow [2]\neg d, [1]\neg r \vee \neg d, d, r \vee \neg d \vdash [2]c$

(c is replaceable and, based on step (6), we take the next relevant clause from the origin-place of address [2].)

and $[1]\neg r \vee \neg d, a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]b$

$\leftrightarrow [1]\neg r \vee \neg d, d, r \vee \neg d \vdash [2]d$ (We used step (4))

and $[1]\neg r \vee \neg d, a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]b$

$\leftrightarrow \bot \vdash \bot$ (Based on rule (R2))

and $[1]\neg r \vee \neg d, a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]b$

$\leftrightarrow \vdash \top$ and $[1]\neg r \vee \neg d, a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]b$

(After applying rule (R1) only proof of the second formula remained.)

$\leftrightarrow [1]\neg r \vee \neg d, a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]b$

(Again, b is replaceable so we go to origin-place of [1] and take the next relevant clause [1] $\neg r \vee \neg d$.)

$\leftrightarrow a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]r$ and

$a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]d$ (Rule (R3))

$\leftrightarrow [3]\neg d, a \vee \neg c, a \vee \neg d, d \vdash \bot$ (Rule (R2))

and $a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]d$

$\leftrightarrow a \vee \neg c, a \vee \neg d, d \vdash [3]d$ (Rule (R3))

and $a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]d$

$\leftrightarrow \bot \vdash \bot$ (Rule (R2))

and $a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]d$

$\leftrightarrow \vdash \top$ (Rule (R1))

and $a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]d$

$\leftrightarrow a \vee \neg c, a \vee \neg d, d, r \vee \neg d \vdash [1]d$

$\leftrightarrow a \vee \neg c, \bot \vdash \bot$ (Rule (R2))

$\leftrightarrow \vdash \top$ (Rule (R1))

QED

We can extend propositional PL proof procedure to the first order case considering formulas with eliminated quantifiers. We use unification algorithm to substitute variables with instances of Herbrand universe. If a disjunction containing variables becomes a relevant clause, we make a copy of it in which variables may gain values. This copy will be further processed. This enables usage of the same disjunction more than once with a different variable values that appear in it.

Example 2. Prove the sequent:

$(**)$ $\neg\alpha(X) \vee \beta(f(X)), \neg\beta(X) \vee \gamma(g(X)), \neg\gamma(X) \vdash \neg\alpha(c)$

Proof.

$(**) \leftrightarrow [1]\beta(f(c)), \neg\alpha(X) \vee \beta(f(X)), \neg\beta(X) \vee \gamma(g(X)), \neg\gamma(X) \vdash \bot$

(We make a copy of disjunction $\neg\alpha(X) \vee \beta(f(X))$ and then apply (R2) on that copy.)

$\leftrightarrow \neg\alpha(X) \vee \beta(f(X)), \neg\beta(X) \vee \gamma(g(X)), \neg\gamma(X) \vdash [1]\beta(f(c))$

$\leftrightarrow [2]\gamma(g(f(c))), \neg\alpha(X) \vee \beta(f(X)), \neg\beta(X) \vee \gamma(g(X)), \neg\gamma(X) \vdash \bot$

(In the first disjunction we also find relation β, but applying rule (R2) on that disjunction gives $\top$, so it will not effect further proving)

$\leftrightarrow \neg\alpha(X) \vee \beta(f(X)), \neg\beta(X) \vee \gamma(g(X)), \neg\gamma(X) \vdash [2]\gamma(g(f(c)))$

$\leftrightarrow \bot, \neg\alpha(X) \vee \beta(f(X)), \neg\beta(X) \vee \gamma(g(X)), \neg\gamma(X) \vdash \bot$

$\leftrightarrow \vdash \top$

QED

4. ABOUT THE PROGRAM ITSELF

Let us describe the program which implements given rules and theorems. To allow an efficient and memory unconstrained implementation we use dynamic structures.

Structure

```
struct tree{
        int inf;
        char *root;
        struct tree *left;
        struct tree *right;};
```

is used for saving and handling disjunctions. Since the algorithm is constantly using a set of disjunctions, from which a formula will be proved, it was natural to utilize structure:

```
struct list{
        struct tree *root;
        struct list *next;};
```

that enables recording of such set, as a list of trees. Because there are variables in formulas, we must use a list in which the variable values will be memorized.

```
struct unif{
        char *var;
        struct tree *repl;
        struct unif *next;};
```

The whole algorithm of PL prover runs using the structure:

```
struct stek{
        int lev;
        int dis;
        struct list *ls;
        struct tree *dr;
        struct stek *next;};
```

This structure saves the clause itself and the set of disjunctions for every relevant clause. The clause will be proved by this saved set. Further more, the structure possesses the information about address, namely level, of relevant clause and the information whether the clause was element of disjunction, or not. The structure also saves a list of variables which gained values during the previous step of proving. It is that previous step where this relevant clause appeared. The structure also has an information about the place in variable list in which a new variable list, directly connected to this element of the structure, should be added.

After starting the program, the very first thing to do is to read input disjunctions separated by commas. Instead of operators $\vee$ and $\neg$, words **or** and **neg** are used. Function **ChrTrLst** puts this disjunction in the list. Function **PL** seeks the formula (literal or negation of the literal) which will be proved based on given disjunctions, reads it using function **ChrTrLst** and starts the main part of PL prover. That is a loop which starts by taking from the structure **struct stek** a relevant clause (with respect to the formula which will be proved), information about it and appropriate list of disjunctions. Proving every relevant clause will differ on the type of the clause. Disjunctions, literals and negations of literals each have separate proving solutions. Every part of disjunction becomes a new relevant clause and will be proved based on the same list of disjunctions. For proving a literal or a negation of a literal appropriate form of rule (R2) is used.

```
void PL(struct list *p) {
  ChrTrLst(&prove);
  while(the proof is not reached){
    if(there are relevand clauses) {
      taking the relevant clauses and the appropriate list of disjunction
      adding variables in variables list
    }
```

```
    if(operator is "or") {
      Dis_Lst(disjunction, unifier list);
      ListRel(main structure, list of parts of disjunction);
    }
    if(operator is "not") {
      relev=Bot(list of disjunction, literal, unifier list,1);
      ListRel(main structure, list of relevant clauses);
    }
    if(we observe a literal ){
      relev=Bot(list of disjunction, literal, unifier list,0);
      ListRel(main structure, list of relevant clauses);
    }
    if(relev==0) {
      checking if the formula is proved
    }
  }
}
```

Fig.1. Function PL

Function **Bot** applies the rule (R2), and returns the list of relevant clauses. Appliance of rule (R2) for propositional case is rather simple. If a disjunction, used for proving of some clause, contains variables, a copy of this disjunction is made, and the rule (R2) is applied on that copy. Before applying the rule (R2) it is required to use unification algorithm to find values for the variables such that given disjunction or a part of it could be unified with a clause that is being proved. All combinations of the values are memorized in separate lists since every combination gives new relevant clauses. Here, just as in PROLOG, a variable can change its value only in the place where it first got it. Hence, it is necessary to memorize each such location. The change of the value is required when prof with the previous value

gave no effect. The list of relevant clauses is taken by the function **ListRel** which makes a new element in the structure **struct stek** for every relevant clause.

```
struct list* Bot(struct list **p, struct tree *s, char *v) {
  while(there are disjunction in the list) {
    if (a literal could be unified with a disjunction or a part of it) {
      if(there are variables in disjunction){
        creating copy of disjunction and adding it to the list of disjunction
      }
      substituting variables from disjunction with appropriate values
            from unifier list
      substituting literal with value v
      finding a value of disjunction
      if(disjunction has changed its value) {
        if(value of disjunction is not "0" or "1") {
          adding disjunction to list of relevant clauses
        }
      }
    }
    if the variables which appear may gain different values we make
          some special elements of the main structure
    taking the next disjunction
  }
  returning list of relevant clauses
}
```

Fig.2. Function BOT

In every iteration the program takes an element from the structure **struct stek** and makes a new list of relevant clauses based on it. If a relevant clause which is

going to be used is a disjunction, then function **ListDis** makes a list of elements of disjunction. This list is then taken by function **ListRel**.

If there are no new relevant clauses in any step, then we should check if in an active list of disjunctions (in any place) contains 0 (which corresponds to $\perp$). There are five different cases that can occur:

1. If in the list there is 0 and in the structure **struct stek** there is no relevant clause which used to be a part of disjunction, then the algorithm stops with the conclusion that the formula is proved, that is that the given formula is logical consequence of the given set of disjunction.

2. If in a list there is 0 and in the structure **struct stek** is an element which used to be a part of disjunction, then every clause in front of the found part of disjunction has to be removed (that is the relevant clauses that appeared later in the list) and the algorithm proceeds.

3. If in a list there is no 0 and the structure **struct stek** is not empty all clauses that were parts of disjunctions should be removed. These clauses should be removed because they used to be parts of unprovable disjunction and by proving them we could come to a wrong conclusion. These clauses are at the beginning of the structure and follow each other in the structure. Removal stops once a clause, which is not a part of disjunction, appears. After this event the algorithm continues.

4. If in a list there is no 0 and structure **struct stek** is not empty, the algorithm continues running.

5. If in a list there is no 0 and in a structure **struct stek** are no more elements algorithm stops with conclusion that the formula is not proved, i.p. that given formula is not logical consequence of the given set of disjunction.

5. TESTING RESULTS

The program described in Section 4 was compiled under Turbo C compiler as 16bit application. We get the following results on PC 133MHh with 32MB RAM:

sequent	time	is proved	steps
$p \vee \neg q, \neg p \vee q \vdash p$	0.00	no	2
$a \vee \neg b \vee \neg c, b \vee \neg c, a \vee \neg d, d \vee \neg e \vdash a$	0.05	no	6
$p \vee \neg a \vee \neg b, a \vee \neg c, a \vee \neg d, d, p \vee \neg r \vee \neg d, r \vee \neg d \vdash p$	0.11	yes	10
$\neg\alpha(X) \vee \beta(f(X)), \neg\beta(X) \vee \gamma(g(X)), \neg\gamma(X) \vdash \neg\alpha(c)$	0.11	yes	3
$\neg p6 \vee \neg p7 \vee \neg p2, \neg p4 \vee p1 \vee p3, p7 \vee \neg p3 \vee p1,$ $\neg p4 \vee p7 \vee p1, \neg p1 \vee p6 \vee p4, \neg p7 \vee p5 \vee \neg p1,$ $p3 \vee p7 \vee p4, p2 \vee p1 \vee \neg p5, p6 \vee \neg p4 \vee \neg p3,$ $p4 \vee \neg p7 \vee \neg p3, \neg p1 \vee p3 \vee p7, \neg p6 \vee p1 \vee \neg p5,$ $p1 \vee p4 \vee p5, p1 \vee \neg p7 \vee \neg p2, \neg p4 \vee p3 \vee \neg p5 \vdash p6$	0.82	yes	42

Table: Some results of testing

The Table shows the results which the program gives with given formulas. The values show time in seconds, in which program finished its work; if the formula is logical consequence of given set of disjunction and the number of steps in proof.

When the program works with sequents consisted of large numbers of disjunction working time will depend on the position of disjunctions in the set. At this point, due to limitations of compiler used, there is a limit to a number of disjunction that can be manipulated. This number depends on actual problem, and is usually around 100.

One of the most famous logical problems unsolvable for PROLOG is: $p \vee \neg q, \neg p \vee q \vdash p$. PROLOG algorithm defines this problem using following clauses $p :- q.\ q :- p.$. Based on this set PROLOG can not prove either p or q. The PL algorithm solves this problem as follow:

$$\begin{aligned} p \vee \neg q, \neg p \vee q \vdash p &\leftrightarrow [1]\neg q, \top \vdash \bot \\ &\leftrightarrow [1]\neg q \vdash \bot \\ &\leftrightarrow \vdash [1]q \\ &\leftrightarrow \vdash \bot \end{aligned}$$

This program completely satisfies PL algorithm described in [2]: it accomplishes both ability to be applied to any set of disjunctions and the relevance of steps.

The algorithm is using Herbrand's universe which allows cases in which algorithm does not stop in predicate case. On the other hand, one of the good sides is that it offers a framework for appliance to some other logical systems, for example for fuzzy logic.

References

[1] C. Chang, R. Lee : *Symbolic logical and mechanical theorem proving*, Moscow, 1983

[2] S. Presić : *How to generalize logic programming to arbitrary set of clauses*, Publications de l'institut mathematique, Belgrade, 61/75(1997) 137-152

[3] S. Presić : *PROLOG relacijski jezik*, Belgrade, 1996