PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

LOGICAL DERIVATION OF A PROLOG INTERPRETER

Kazuhiro Fuchi

ICOT Research Center
Institute for New Generation Computer Technology
Tokyo, Japan

ABSTRACT

This paper describes the derivation of a Prolog interpreter whose principal control structure involves the transformation of logical formulas. A variable for representing "continuation" has been added to the basic predicates used in the specifications for the Prolog interpreter. The AND operator has been eliminated by the introduction of this variable. The OR operator has been similarly replaced.

Consequently, a logical formula corresponding to a sequential interpreter with two kinds of stacks is derived by the side effects of AND and OR elimination. A tail recursion optimization technique can be easily and naturally introduced for this interpreter. The derived program can be easily converted into a conventional programming language, in part, because it contains a component analogous to the conventional if-then-else statement.

1 *AND* ELIMINATION, *OR* ELIMINATION

The sequential Prolog interpreter is essentially a program for traversing AND/OR trees depth-first. AND/OR tree manipulation is sufficient to demonstrate the methodology used in this paper; therefore, all but the essential points have been omitted for the sake of simplicity.

The following definitions apply to the AND/OR trees:

- An AND node is represented by a list of goals in which each goal is denoted by a list of clauses.
- An empty list for the AND node is interpreted as logically true.
- An OR node is represented by a list of clauses in which each clause is denoted either by a list of goals or by an empty list.
- An empty list for the OR node is interpreted as logically false.

This data structure is provided only for the sake of simplifying the following description, and to avoid the use of labeled AND/OR trees. The non-labeled AND/OR tree introduced in this paper is equivalent to a labeled AND/OR tree.

The predicate *prove* is introduced for this AND/OR tree. Its sole argument is the list of goals, in other words, the AND node. When *prove(X)* succeeds, it means that all the goals in the list X are proved true.

The predicate *try* is also introduced for the list of clauses. *try(X)* represents the result of ORing clauses.

The AND/OR tree traversal is specified by means of the *prove* and *try* predicates.

```
prove([]) :- true.
prove([Goal|Goals]) :-
        try(Goal), prove(Goals).
try([]) :- fail.
try([Clause|Clauses]) :-
        prove(Clause) ; try(Clauses).
```

The above formula can be understood as a nondeterministic traversal program. In the current discussion, however, this formula is regarded as the "specification" of the AND/OR tree traversal and is used for the derivation explained below.

A comment is in order on the notation throughout this paper. "Logical implication" is represented in inverse order by the Prolog symbol for implication (:-). "Logical and" is denoted by a comma (,), and "logical or" by a semicolon (;). Note that these notational conventions are not part of the normal Prolog syntax but are used only to represent logical formulas. A specific control structure is not assumed here.

1.1 AND elimination

The predicate *and_prove* is introduced for the list of goal lists. It is defined as follows:

```
and_prove([]) ≡ true.
and_prove([Goals|Goals_list])
      ≡ prove(Goals), and_prove(Goals_list).
```

The extended *prove* and *try* predicates are then defined as shown below. Note that the auxiliary argument is introduced and the suffix "1" is attached to distinguish an extension.

```
prove_1(A,B) ≡ prove(A), and_prove(B).
try_1(A,B) ≡ try(A), and_prove(B).
```

When the auxiliary variable in the above definitions becomes an empty list, the extended definitions are

reduced to the original definition.

```
prove_1(A,[]) ≡ prove(A), and_prove([]).
              ⇒ prove(A), true.
              ⇒ prove(A).

try_1(A,[]) ≡ try(A), and_prove([]).
            ⇒ try(A), true.
            ⇒ try(A).
```

The properties of the extended predicates are derived from the following transformations:

(i) In the case of *prove(empty_list, empty_list)*

```
prove_1([],[]) ≡ prove([]), and_prove([]).
               ⇒ true, true.
               ⇒ true.
```

(ii) In the case of *prove(empty_list, list)*

```
prove_1([],[A|B]) ≡ prove([]), and_prove([A|B]).
                  ⇒ true, and_prove([A|B]).
                  ⇒ and_prove([A|B]).
                  ⇒ prove(A), and_prove(B).
                  ⇒ prove_1(A,B).
```

(iii) In the case of *prove(list, list)*

```
prove_1([A|B],C) ≡ prove([A|B]), and_prove(C).
                 ⇒ try(A), prove(B), and_prove(C).
                 ⇒ try(A), and_prove([B|C]).
                 ⇒ try_1(A,[B|C]).
```

(iv) In the case of *try(empty_list, list)*

```
try_1([], A) ≡ try([]), and_prove(A).
             ⇒ fail, and_prove(A).
             ⇒ fail.
```

(v) In the case of *try(empty_list, empty_list)*

```
try_1([A|B],C) ≡ try([A|B]), and_prove(C).
               ⇒ (prove(A) ; try(B)),
                 and_prove(C).
               ⇒ (prove(A), and_prove(C)) ;
                 (try(B), and_prove(C)).
               ⇒ prove_1(A,C) ; try_1(B,C).
```

In summary, the following interpreter is derived:

```
prove_1([],[]) :- true.
prove_1([],[Goals|Remaining_goals]) :-
        prove_1(Goals,Remaining_goals).
prove_1([Goal|Goals],Remaining_goals) :-
        try_1(Goal,[Goals|Remaining_goals]).
try_1([],Remaining_goals) :- fail.
try_1([Clause|Clauses],Remaining_goals) :-
        prove_1(Clause,Remaining_goals) ;
        try_1(Clauses,Remaining_goals).
```

Fig. 1 The derived program with AND elimination

This derived program shows the result of AND elimination. The auxiliary variable contains the list of goal lists which were unaffected by the left-to-right depth-first expansion of the AND/OR tree, as shown in Fig. 2. This variable is used to represent the concept of continuation as defined in functional programming (Carlsson 1984) and to represent the invocation stack of conventional programming languages. Moreover, the second formula in Fig. 1 is interpreted as the "pop-up" stack operation, and the third formula as the "push-down" stack operation.

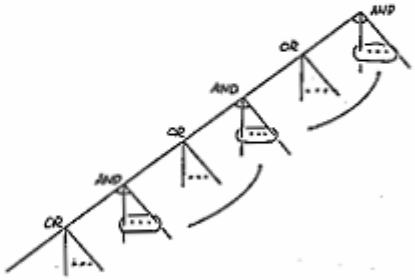

Fig. 2 The expansion of AND/OR trees

Because of the elimination of AND, the formulas in Fig. 1 consist only of OR operations. Therefore Fig. 1 can be regarded as a program for an OR-parallel machine. Moreover, it is easy to rewrite the program as a kind of dataflow graph (see Fig. 3) representing pipelined processing.

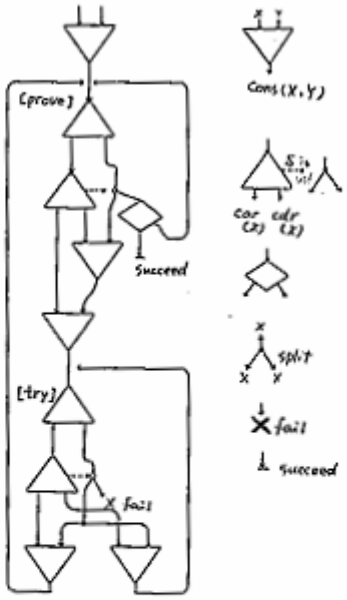

Fig. 3 A dataflow graph of a program from which AND has been eliminated

1.2 Elimination of OR

Since AND and OR are dual, OR can be similarly eliminated.

```
or_try([]) ≡ fail.
or_try([Clause|Clauses]) ≡
        try(Clause) ; or_try(Clauses).
```

Fig. 4 OR processing definition

```
prove_2(A,B) ≡ prove(A) ; or_try(B)
try_2(A,B) ≡ try(A) ; or_try(B)
```

Fig. 5 Predicates for OR elimination

```
prove_2([],Alternatives) :- true.
prove_2([Goal|Goals],Alternatives) :-
        try_2(Goal, Alternatives),
        prove(Goals,Alternatives).
try_2([],[]) :- fail.
try_2([],[Clauses|Alternatives]) :-
        try_2(Clauses,Alternatives).
try_2([Clause|Clauses],Alternatives) :-
        prove_2(Clause,[Clauses|Alternatives]).
```

Fig. 6 AND/OR tree program with OR elimination

The above program is deterministic. The auxiliary variable, in this case, represents the backtrack stack. The entire program can easily be rewritten in a functional language.

1.3 Elimination of AND-OR

OR elimination can be applied to the formulas in Fig. 1, which are the result of AND elimination. The following are the auxiliary and extended predicates for applying the elimination:

```
and_or_prove([]) ≡ true.
and_or_prove([[Clause,Goals]|Remainings])
        ≡ prove_2(Clause,Goals),
          and_or_prove(Remainings).
```

Fig. 7 AND-OR processing definition

```
prove_3(Clause,Remainings,Alternatives)
        ≡ prove_2(Clause,Alternatives),
          and_or_prove(Remainings).
try_3(Clause,Remainings,Alternatives)
        ≡ try_2(Clause,Alternatives),
          and_or_prove(Remainings).
```

Fig. 8 New predicates for AND-OR elimination

A procedure similar to that described in Section 1.2 obtains the following result:

```
prove_3([],[],D) :- true.
prove_3([],[[A,B]|C],D) :- prove_3(A,C,B).
prove_3([A|B],C,D) :- try_3(A,[[B,D]|C],D).
try_3([],A,[]) :- fail.
try_3([],A,[B|C]) :- try_3(B,A,C).
try_3([A|B],C,D) :- prove_3(A,C,[B|D]).
```

Neither AND nor OR appear on the right hand side of the above formulas. Notice that a pointer to the call stack is placed on the backtrack stack. The above program can be translated into either a Single Assignment Language (SAL) program like LUCID (Ashcroft 1982) or a conventional language program. Since the program is tail recursive, the result of the translation would be a simple loop program.

1.4 Elimination of OR-AND

Notice also that we can change the order of elimination, i.e., we can eliminate OR before AND. The results of this transformation are shown below:

```
or_and_try([]) ≡ fail.
or_and_try([[Goal,Goals]|Remainings])
        ≡ try_1(Goal,Goals),
          or_and_try(Remainings).
```

Fig. 9 OR-AND processing definition

```
prove_4(Clause,Remainings,Alternatives)
        ≡ prove_1(Clause,Remainings) ;
          or_and_try(Alternatives).
try_4(Clause,Remainings,Alternatives)
        ≡ try_1(Clause,Remainings) ;
          or_and_try(Alternatives).
```

Fig. 10 New predicates for OR-AND elimination

```
prove_4([],[],D) :- true.
prove_4([],[A|B],C) :- prove_4(A,B,C).
prove_4([A|B],C,D) :- try_4(A,[B|C],D).
try_4([],A,[]) :- fail.
try_4([],A,[[B,C]|D]) :- try_4(B,C,D).
try_4([A|B],C,D) :- prove_4(A,C,[[B,C]|D]).
```

Fig. 11 AND/OR tree program with OR-AND elimination

The derived programs for AND-OR elimination and OR-AND elimination are essentially the same. However, the stack structure differs in each case. In the Prolog interpreter derivation, the problem of saving the environment also causes considerable differences.

1.5 Tail Recursion Optimization

Consider a case in which the first argument on the left hand side of the third formula in Fig. 1 contains only one element. In this case, stack manipulation can be omitted, instead of pushing and poping the empty list on and off the call stack.

```
prove_1([A],B)
        ≡ try_1(A,[[]|B]).
        ≡ try(A), and_prove([[]|B]).
        ≡ try(A), prove([]), and_prove(B).
        ≡ try(A), and_prove(B).
        ≡ try_1(A,B).
```

This saves space for the call stack, and time for stack manipulation. This is a well-known technique for Prolog compilers and interpreters.

2 THE DERIVATION OF PROLOG INTERPRETER

The Prolog interpreter is derived directly from the AND/OR tree traversing program discussed above, although some additional techniques are needed to deal with variable binding and other techniques concerned with environment handling. The specification is given here for (pure) Prolog along with variables representing the environment:

```
prove([],Goal,Env) :- fail.
prove(Env,[],Env).
prove(Env,[Goal|Goals],NewEnv) :-
        clauses(Goal,Clauses),
        try(Env,Goal,Clauses,IntEnv),
        prove(IntEnv,Goals,NewEnv).
try(Env,Goal,[],NewEnv) :- fail.
try(Env,Goal,[(Head:-Body)|Clauses],NewEnv) :-
        unify(Env,Goal,Head,IntEnv),
        prove(IntEnv,Body,NewEnv) ;
        try(Env,Goal,Clauses,NewEnv).
```

Fig. 12 Specification of Prolog interpreter

The predicate *clause* gathers clauses whose predicate names are the same with that of *Goal*. The result is included in the list *Clauses*. The predicate *unify* unifies *Goal* and *Head*. The environment changes from *Env* to *IntEnv*. If *Goal* and *Head* fail to unify, *unify* returns the empty-list as *IntEnv*.

```
prove_list(E,[],E) ≡ true.
prove_list(E,[Gs|C],NE)
        ≡ prove(E,Gs,IE),
          prove_list(IE,C,NE).
prove_1(E,Gs,Co,NE)
        ≡ prove(E,Gs,IE),
          prove_list(IE,Co,NE).
try_1(E,G,Cl,Co,NE)
        ≡ try(E,G,Cl,IE),
          prove_list(IE,Co,NE).
```

Fig. 13 Definition for AND elimination

```
prove_1(E,[],[],E).
prove_1(E,[],[Gs|Co],NE) :-
        prove_1(E,Gs,Co,NE).
prove_1(E,[G|Gs],Co,NE) :-
        clauses(G,Cl),
        try_1(E,G,Cl,[Gs|Co],NE).
try_1(E,G,[(H:-B)|Cl],Co,NE) :-
        unify(E,G,H,IE),
        prove_1(IE,B,Co,NE) ;
        try_1(E,G,Cl,Co,NE).
```

Fig. 14 Result of AND elimination

```
try_list([e(E,G,C,S)|B], NE)
        ≡ try_1(E,G,C,S,NE) ;
          try_list(B,NE).
prove_2(E,G,S,B,NE)
        ≡ prove_1(E,G,S,NE) ;
          try_list(B,NE).
```

```
try_2(E,G,C,S,B,NE)
        ≡ try_1(E,G,C,S,NE) ;
          try_list(B,NE).
```

Fig. 15 Definition for AND-OR elimination

```
prove_2([],G,S,[e(OE,OG,OC,OS)|B],NE)
        ≡ try_1([],E,G,S,NE) ;
          try_list([e(OE,OG,OC,OS)|B],NE).
        ⇒ try_list([e(OE,OG,OC,OS)|B],NE).
        ⇒ try_2(OE,OG,OC,OS,B,NE).
prove_2(E,[],[G|S],B,NE)
        ≡ prove_1(E,[],[G|S],NE) ;
          try_list(B,NE).
        ⇒ prove_1(E,G,S,NE) ;
          try_list(B,NE).
        ⇒ prove_2(E,G,S,B,NE).
prove_2(E,[G|Gs],S,B,NE)
        ≡ prove_1(E,[G|Gs],S,NE) ;
          try_list(B,NE).
        ⇒ clauses(G,Cs),
          try_1(E,G,Cs,[Gs|S],NE) ;
          try_list(B,NE).
        ⇒ clauses(G,Cs),
          try_2(E,G,Cs,[Gs|S],B,NE).
try_2(E,G,[],S,[e(OE,OG,OC,OS)|B],NE)
        ≡ try_1(E,G,[],S,NE) ;
          try_list([e(OE,OG,OC,OS)|B],NE).
        ⇒ try_2(OE,OG,OC,OS,B,NE).
try_2(E,G,[(H:-B)|C],S,BS,NE)
        ≡ try_1(E,G,[(H:-B)|C],S,NE) ;
          try_list(BS,NE).
        ⇒ unify(E,G,H,E1),
          prove_1(E1,B,S,NE) ;
          try_1(E,G,C,S,NE) ;
          try_list(BS,NE).
        ⇒ unify(E,G,H,E1),
          prove_1(E1,B,S,NE) ;
          try_list([e(E,G,C,S)|BS],NE).
        ⇒ unify(E,G,H,E1),
          prove_2(E1,B,S,[e(E,G,C,S)|BS],NE).
```

Fig. 16 Elimination process

```
prove_2([],G,S,[e(OE,OG,OC,OS)|B],NE) :-
        try_2(OE,OG,OC,OS,B,NE).
prove_2(E,[],[],BT,E).
prove_2(E,[],[Gs|Co],BT,NE) :-
        prove_2(E,Gs,Co,BT,NE).
prove_2(E,[G|Gs],Co,BT,NE) :-
        clauses(G,Cl),
        try_2(E,G,Cl,[Gs|Co],BT,NE).
try_2(E,G,[],Co,[e(OE,OG,OCl,OCo)|BT],NE) :-
      try_2(OE,OG,OCl,OCo,BT,NE).
try_2(E,G,[(H:-B)|Cl],Co,BT,NE) :-
      unify(E,G,H,IE),
      prove_2(IE,B,Co,[e(E,G,Cl,Co)|BT],NE).
```

Fig. 17 Result of AND-OR elimination

In the derivation of the formulas shown in Fig. 16, it should be noted that OR and AND are exchanged by the distributive law. Also, variable names must be used consistently. Special care is called for in the use of the *clauses* and *unify* predicates. These predicates here are assumed to be always successful.

The formulas shown in Figures 12, 14 and 16, can be treated as an entire Prolog program. The result of execution of this program is given in Appendix.

3 CONCLUSION

Though there have been other attempts to derive Prolog interpreters from specifications (Carlsson 1984); (Furukawa 1982); (van Emden 1981), the method proposed here is far simpler. The operations on call stacks and backtrack stacks by the method described in this paper constitute a significant factor in this simplification.

The derived interpreter is sound because the transformation method is essentially equivalent to logical transformation of propositions (Tamaki 1984). Thus, derived formulas and programs can be verified from the original specification. However, the interpreter cannot be said to be complete, because the depth-first algorithm, by nature, does not assure termination.

The logical derivation of programs appears to hold some promise (Scherlis 1983). However, as there seems to be no general derivation rule, it is necessary to find heuristics. For this reason, any attempt at logical program transformation is to be encouraged in the hope that broadly applicable heuristics will eventually be developed. The methodology employed in this paper, though it deals only with the derivation of known algorithms, can be viewed as one of attempt. As another example, it might prove interesting to apply the ideas presented in this paper to the field of arithmetic simplification. Note that predicate extention constitutes one of the heuristics mentioned above.

4 ACKNOWLEDGEMENT

The author is grateful to his young colleagues in ICOT, who are prone to argue over various aspects of this work. Special thanks are due to T. Kurokawa, K. Sakai, Y. Tanaka and H. Yasukawa, who helped in preparing the printed output, and creating the sample programs in Appendix.

REFERENCES

Ashcroft, E. A. and Wadge, W. W., *A summary of LUCID for programmers* (1981 version), Univ. of Waterloo, CS-82-57, 1982.

Carlsson, M., *On implementing Prolog in functional programming*, in Proc. 1984 International Symposium on Logic Programming, pp.154-159, IEEE, 1984.

Furukawa, K., Nitta, K. and Matsumoto, Y., *Prolog interpreter based on concurrent programming*, in Proc. 1st Int. Conf. Log. Prog., pp.38-44, 1982.

Scherlis, W. L. and Scott, D. S., *First steps towards inferential programming*, IFIP-83, pp.199-212, 1983.

Tamaki, H., *Semantics of a logic programming language with reducibility predicate*, in Proc. 1984 International Symposium on Logic Programming, pp.259-264, IEEE, 1984.

van Emden, M. H., *An algorithm for interpreting Prolog program*, Univ. of Waterloo, 1981.

APPENDIX

```
/* Prolog Interpreter written in DEC-10 Prolog */

/* The variables representing the environment */
/* do not appear in the program, because they */
/* are implicitly supported by the execution */
/* mechanism of Prolog system.  */

/* Specification Program */

prove([]).
prove([Goal|Goals]) :-
        clauses(Goal,Clauses),
        try(Goal,Clauses),
        prove(Goals).
try(Goal,[(Head:-Body)|Clauses]) :-
        unify(Goal,Head), prove(Body) ;
        try(Goal,Clauses).

/* Program from which AND has been eliminated */

/* The third clause of prove1 is for */
/* tail recursion optimization.  */

prove1([],[]).
prove1([],[Goal|Cont]) :-
        prove1(Goal,Cont).
prove1([Goal],Cont) :- !,
        clauses(Goal,Clauses),
        try1(Goal,Clauses,Cont).
prove1([Goal|Goals],Cont) :-
        clauses(Clauses),
        try1(Goal,Clauses,[Goals|Cont]).
try1(Goal,[(Head:-Body)|Clauses],Cont) :-
        unify(Goal,Head), prove1(Body,Cont) ;
        try1(Goal,Clauses,Cont).

/* Program from which both AND and OR have */
/* been eliminated */

/* The third clause of prove2 and */
/* the second clause try2 are for */
/* tail recursion optimization.  */

prove2([],E,[],BT) :-
        write_answer(E).
prove2([],E,[G|Co],BT) :-
        prove2(G,E,Co,BT).
prove2([G],E,Co,BT) :- !,
        clauses(G,Cl),
        try2(G,E,Cl,Co,BT).
prove2([G|Gs],E,Co,BT) :-
        clauses(G,Cl),
        try2(G,E,Cl,[Gs|Co],BT).
try2(G,E,[],Co,[e(OG,OE,OCl,OCo)|BT]) :-
        try2(OG,OE,OCl,OCo,BT).
try2(G,E,[(H :- B)],Co,BT) :- !,
        unify(G,H),
        prove2(B,E,Co,BT).
try2(G,E,[(H :- B)|Cl],Co,BT) :-
        copy_term(e(G,E,Co),e(G,E,C)),
        unify(G,H),
        prove2(B,E,C,[e(G,E,Cl,Co)|BT]).

/* UTILITIES */

write_answer(Answer) :-
        write(' '),
        write(Answer).
solve(DB,Goal):-
        abolish(db,1),
        assert(db(DB)),
        prove(Goal),
        write_answer(Goal).
solve1(DB,Goal):-
        abolish(db,1),
        assert(db(DB)),
        prove1(Goal,[]),
        write_answer(Goal).
solve2(DB,Goal):-
        abolish(db,1),
        assert(db(DB)),
        prove2(Goal,Goal,[],[]).
select_clause([C|DB],[(H:-B)|DBc],G,Gc,[C|Cl]) :-
        unify(H,Gc), !, copy_term(G,Gc0),
        select_clause(DB,DBc,G,Gc0,Cl).
select_clause([_|DB],[_|DBc],G,Gc,Cl) :- !,
        select_clause(DB,DBc,G,Gc,Cl).
select_clause([],[],G,Gc,[]).
clauses(G,Cl) :-
        db(DB), db(DBc), copy_term(G,Gc),
        select_clause(DB,DBc,G,Gc,Cl).
```