:
:

DO LOGIC PROGRAMS RESEMBLE PROGRAMS IN CONVENTIONAL LANGUAGES ?

Włodzimierz Drabent

Department of Computer and Information Science
Linköping University
581 83 Linköping, Sweden

present address: Institute of Computer Science
Polish Academy of Sciences
P.O.Box 22, 00-901 Warszawa PKiN, Poland

This is a reformated version of the paper that appeared in "Proceedings of 1987 Symposium on Logic Programming". The author's e-mail is `wdr@ida.liu.se` and the homepage is `http://www.ipipan.waw.pl/~drabent/`. The second paper-mail address above is obsolete.

ABSTRACT: It was suggested by Mellish that “many Prolog programs are not radically different in kind from programs written in conventional languages”. This paper attempts to formalize the concept of a “conventional” logic program. An experiment was performed to check how often Prolog programs fall in our restricted class of conventional programs and ascertain which programming techniques result in programs falling outside this class. For the experiment a sample of Prolog programs was selected ranging from simple student programs to a 52K character data base implementation. The results confirm Mellish's hypothesis. On the other hand the experiment disclosed some programming techniques which are unique for logic programming.

1. INTRODUCTION

It was suggested by Mellish [M] that “many Prolog programs are not radically different in kind from programs written in conventional languages”. We share this opinion and we believe that a better understanding of “conventional” and “non-conventional” techniques of logic programming is essential for improving efficiency of implementations. This paper attempts to formalize the concept of a “conventional” logic program and describes an experiment based on this formalization. The aim of the experiment is to check how often Prolog programs fall in our restricted class of conventional programs and ascertain which programming techniques result in programs falling outside this class.

Our intuition of conventional programs is similar to Mellish's *directional programs* [M] and close to Kluźniak's ground Prolog [K]. It is formalized as a restriction with regard to data flow during computation. The restriction originates from [DtM] and was used in [DiM] as a basis for a model of AND-parallel computations.

For the experiment a sample of Prolog programs was selected ranging from simple student programs to a 52K character data base implementation. For checking whether a given program satisfies our restrictions a program in Prolog was developed. The results of the experiment confirm Mellish's hypothesis. About half of the programs have satisfied the restrictions; substantial parts of the remaining ones were ”conventional” too. On the other hand the experiment disclosed some programming techniques which are unique for logic programming.

In the next section of this paper a class of *simple logic programs* is introduced and their properties are discussed. The third section describes a method which was used to check whether a program is simple and summarizes the results of the experiment. The fourth section presents programming techniques which led to some of the sample programs being classified as not being simple.

2. SIMPLE LOGIC PROGRAMS

Our intuition of "conventional" computation assumes transformation of input data into output data. When applied to logic programs, this would mean that the argument positions of every predicate symbol are divided into input positions and output positions. Following [DtM] we call such programs *annotated logic programs*. Furthermore, according to the model-theoretic semantics of logic programs [AE], input and output data are ground terms. Thus, we want to focus our attention on annotated logic programs which transform input ground terms into output ground terms. Notice that this does not exclude non-determinism; such programs may also fail or produce more than one result. Generally this property of an annotated logic program is undecidable but following [DtM] we give a sufficient condition for it. The condition is based on the observation that nodes of a proof tree of a logic program communicate only via shared variables.

In this paper it is assumed that programs are executed using the Prolog computation rule (the leftmost subgoal becomes a selected goal). Let C be a clause of an annotated program. An occurrence of a variable V in an input position of its head or in an output position of a literal of its body is called a *defining occurrence* of V in C.

Now we require

CONDITION 1

For every clause C of an annotated logic program (including its goal clause)

1. every variable V of C has a defining occurrence in C, and

2. every non-defining occurrence of V in the body of C is preceded by a defining occurrence of V in another literal of C. □

An annotated program satisfying Condition 1 will be called a *simple logic program*. It is easy to show that if a simple logic program is called with a goal clause satisfying Condition 1 then

PROPERTY 1

1. At every step of its computation all input positions of its actual subgoal are ground.

2. When the subgoal succeeds, all its output positions are ground. □

Thus, if the computation succeeds all the positions of the goal clause are ground.

A well known example of a simple logic program is the append program

```
append([ ],L,L).
append([H|L1],L2,[H|L]) :- append(L1,L2,L).
```

where the first two positions of the predicate *append* are input positions and the last one is an output position (which we denote by $append(\downarrow, \downarrow, \uparrow)$). This means that the program is used to obtain the concatenation of two given lists. Another example is the same program with an annotation $append(\uparrow, \uparrow, \downarrow)$ or with $append(\downarrow, \downarrow, \downarrow)$.

It is worth noticing how our input and output positions relate to the modes suggested by Mellish [M] (which are an extension of modes introduced by Warren [W1]). During the computation of a simple logic program the input positions of the actual goal are always instantiated to ground terms, their mode is $++$. On the other hand, the output positions of the actual goal may, or may not, be instantiated so any mode is possible (which is expressed as ? mode). A more detailed discussion on modes appears at the end of this section.

A very similar class of programs is introduced by Bruynooghe [B] to show the possibility of a much more efficient implementation of a restricted logic programming language. The difference between his language and the class of simple programs is that committed choice computation is assumed for the former. For the committed choice computation the set of answers is only a subset of those which can be produced using Prolog-like computation. A variant of that language is ground Prolog [K] introduced for a purpose of presenting a powerful method of type derivation. The difference between simple programs and ground Prolog is that the second uses delayed unification on output positions. This means that at a procedure call only the input positions are unified. Unification of the output positions is done on success of the call. Another related work is [R]. In the terminology of [R] simple logic programs are those for which there exists a definite acyclic well-moding where the dependency relation for each clause coincides with the order of literals.

Although implementation issues are outside of the scope of this paper, it may be expected that at least some of the results of [B] can be used to make implementation of simple logic programs more efficient than in the case of Prolog. It may also be expected that some of the results of [K] can be adapted for simple logic programs.

Simple logic programs have a number of interesting properties:

Every Turing machine can be simulated by a simple logic program [DtM].

A simple logic program can be viewed as a one-sweep attribute grammar [DtM].

A nontrivial subclass of simple logic programs can be executed by a proof procedure employing one-way pattern matching instead of unification [MK].

If the delayed unification on output positions is applied then all simple logic programs can be executed using pattern matching instead of unification. It should be noted that in some cases delaying the unification may itself provide gain in efficiency (in case of failure it is not performed); in some others it may cause unnecessary backtracking (when the unification performed at the moment of procedure call would fail).

Simple logic programs can be executed without occur check. This fact is proved in the Appendix 1.

Data flow analysis of simple logic programs can be done in compile time, thus allowing a simplified model of AND-parallelism [DiM].

A simple method of mode analysis not requiring abstract interpretation is possible (see Appendix 2).

Simple logic programs can be transformed into functional ones using the method described in [R].

3. THE EXPERIMENT

A sample of Prolog programs was tested for being simple. That means it was checked whether for a given program there exists an annotation under which the program is simple. A program called CHECKER was written for this purpose. The program follows the above definition of simple programs but it has to deal with programs which are not pure logic programs. It uses information about annotation of some Prolog-10 standard predicates (eg. *read*, *write*, *is*). For some others, an extension of the above definition would be needed. Such extension seems in many cases very difficult (eg. *assert*, *retract*). It was done only for *;* .

The algorithm used by CHECKER is similar to that presented by Franzen and Hoffman [FH]. Their paper solves a similar problem of assigning directions to extended affix grammars. CHECKER finds all the annotations under which the program is simple. To represent a set of annotations it uses an expression of propositional calculus. Such expressions are built out of propositional variables and connectives. Each position of every predicate symbol has its corresponding propositional variable. The value *false* of the variable means that the related position is input, the value *true* means output.

CHECKER makes one pass over an input program. It assumes that the program is simple and looks for an appropriate annotation. For each clause and each variable occurring in it an expression is built. It represents the fact that, in order for the program to be simple, the variable should occur in an input position in the head or only in output positions of the first literal in which it occurs in the body. For instance, the expression for the clause

```
p(X,Y,X) :- q(Y), r(X,Y,X).
```

and Y is $\neg p.2 \vee q.1$ (where $p.i$ corresponds to the i-th position of p). The expression for X and the same clause is $\neg p.1 \vee \neg p.3 \vee (r.1 \wedge r.3)$. The expressions for all the variables in all the clauses are transformed to conjunctive normal form and conjuncted together. The result is simplified. The simplification is based on the repeating application of two rules: $(a \vee b \vee ...) \wedge a \rightarrow a$ and $(a \vee b \vee ...) \wedge \neg a \rightarrow (b \vee ...) \wedge \neg a$. In the terminology of [FH], they are tautology deletion and unit clause extracture. The resulting expression describes all the annotations under which the given program is simple.

If the expression is unsatisfiable then the program is not simple under any annotation. Such a fact is usually determined by encountering a contradiction (like $a \wedge ... \wedge \neg a$) during simplification. Otherwise the resulting expression can be checked for satisfiability and the possible annotations can be generated (this check usually succeeds, all the non-simple programs found caused contradiction during simplification). Usually, there is only one annotation possible for most of the predicate positions in a typical program.

As programs are usually submitted without their goals, it remains to check whether the intended form of a goal is compatible with (an annotation described by) the resulting expression. (This means checking if Condition 1 is satisfied for the intended goals.) If so then the program is (intended to be used as) a simple logic program.

Consider a standard example:

```
append([ ],L,L).
append([H|L1],L2,[H|L]) :- append(L1,L2,L).
```

The expression for the first clause is $(\neg append.2 \vee \neg append.3)$. The expressions for the second one and H, $L1$, $L2$ and L are, respectively, $(\neg append.1 \vee \neg append.3)$, $(\neg append.1 \vee append.1)$, $(\neg append.2 \vee append.2)$, $(\neg append.3 \vee append.3)$. After conjunction and simplification, the resulting expression $(\neg append.2 \vee \neg append.3) \wedge (\neg append.1 \vee \neg append.3)$ is obtained. It says that the program is simple if the third position of *append* is input (and the remaining two are arbitrary) or if the first two positions are input (and the third arbitrary). If the *append* procedure is a part of a bigger program then some of these possible annotations can be excluded. (End of example)

It is rather difficult to find a good set of benchmark programs. As a first sample we took 23 programs. The first ones (EX7) are seven simple beginner's exercises. Then there is Warren's Prolog benchmark (WPB - 5 programs). SENTENCE is a program translating English sentences into logic formulae. It is taken from the Prolog-20 manual [BPW]. SATISF are two exercise programs which check the satisfiability of a propositional calculus expression. One of them experiments with a kind of intelligent backtracking using non-logical features of Prolog. STORY is a student story generation program, SQUARES a student puzzle solving program and ANSWER is a student story understanding and answering questions program. WARPLAN is a well-known planning program [W][KS] (the STRIPS problem is included). EDITOR is the program described in [KM]. INTERFACE is the user interface of Toy-Prolog (Appendix A.4 of [KS] without the library and the translator). The biggest program analyzed was a data base SPOQUEL [GKS]. CHECKER itself was also included in the sample.

The programs were usually submitted to CHECKER without their goals. The annotations obtained from the checking were compared with the expected goals. If no annotations existed under which a given program was simple, the reason was masked in order to process the remaining part of the program. This allowed us to find more then one such cause in a program.

CHECKER reports the nonexistence of a required annotation after processing the procedure which caused contradiction. It also reports on which predicate position the contradiction occurred. However, the very cause of the contradiction is usually in another place of the program and must be found heuristicly by a human. CHECKER produces output which helps in this task. Actually, the notion of "cause of non-simplicity", being not formally defined, relies on human understanding of a program.

Among 23 programs checked 10 were found to be non-simple, seven of them on the basis of one cause. They are: one program from EX7, one from WPB, SENTENCE, one program from SATISF, STORY, ANSWER, EDITOR. The programs WARPLAN, INTERFACE and SPOQUEL contained several causes of "non-simpleness".

4. REASONS FOR PROGRAMS NOT BEING SIMPLE

6 cases of violating the definition of simple programs were found.

1. Always failing clause.

Such clauses are sometimes used in Prolog programs for their side effects. 24 occurrences were encountered in SPOQUEL, EDITOR, INTERFACE and SATISF. All of them were last clauses of a procedure. A single procedure containing an always failing clause was the only reason for "non-simpleness" of EDITOR and SATISF. The typical form of such a clause is

```
p( Par, _ ) :- error( Par ).
```

where *error* is a procedure which eventually fails. Procedure p is always invoked with its first argument instantiated to a ground term. The first position of p is input but the second one plays the role of an output parameter. Of course a program with such a clause is not simple but it has the Property 1 of simple programs.

2. Insignificant value.

A variable remains uninstantiated while its value is insignificant. An example is

```
..., read( Term, S ), exec( Term, S ), ...
```

Sometimes *read* succeeds with S uninstantiated but then *exec* does not depend on the value of S (the corresponding clauses for *exec* have an anonymous variable as the second argument). The meaning of the program would be equivalent if *read* returned S bound to any ground term. Two such cases were found in INTERFACE.

3. Multidirectional use of a procedure.

A classic example is the procedure *append* which can be used not only to concatenate lists but also to split a list into two. Three cases of "bi-directional" procedures were found. Two of them were single procedures in WARPLAN (procedures *add* and *always*). The third one is more interesting as it concerns a major part of the program ANSWER. A part of the program is used to parse and translate a text to an internal representation. But it is also used to translate such internal representation to its text counterpart. (Actually it required the use of the built-in procedure *var* in one procedure of the program to recognize in which direction the procedure is actually used. Then different clauses of the procedure deal with different directions.)

Multidirectional programs can be transformed to simple ones: multidirectional procedures can be copied, each copy having a new name and a different annotation. It should be noted that in a simple program multidirectionality may still exist on output positions. A current goal may have both an uninstantiated variable or a term in such a position.

4. Two-directional position.

In this case a current goal has a nonground term as an argument. A ground subterm of it plays the role of an input parameter while a variable subterm plays the role of an output one. We can distinguish two subcases.

4a. Glued arguments.

Let us consider the example

```
p(...,f(X,g)) :- ... .
          ... :- p(...,f(h,Y)).
```

This may be rewritten by splitting the last argument of p into two positions. In this way we obtain a simple program by increasing the number of positions of p. We found one example of glued arguments in a predicate with 5 positions.

4b. Goal as an argument.

Another example of two-directional positions that we encountered was passing a whole goal as an argument to a procedure. The purpose of this procedure was to execute the goal in a special way. An example of such a procedure is

```
once( P ) :- P, !.
```

Of course, such a procedure was not pure (first-order) logic program. Three such causes of "non-simpleness" were found in SPOQUEL and INTERFACE. This case may be transformed to a simple program e.g. by writing a new version of this procedure for every such goal occurring in a program.

5. Program bugs.

Two causes of "non-simpleness" appeared to be program bugs.

6. Variable as data.

The case concerns passing a nonground term as an argument of a current goal on a position which is understood as input. From the operational point of view, nonground terms are data of the program. They may also appear in the final results of computations.

This is different not only from simple programs but also from the cases 1, 2, 3 and 4 above where programs can be understood as processing ground terms.

6a. Variables as unique objects.

Variables are used instead of a potentially infinite set of distinct atoms. They never become instantiated.

Example (based on the program SENTENCE). The predicate *sentence* generates predicate calculus formulas with free variables taken from a given list.

```
sentence(_,false).
sentence(_,true).
...
sentence(Freevar, forall(X,S)) :-
      sentence([X|Freevar], S)                    (*)
```

An example goal is $?-\ sentence([\ ], Result)$. The program must be able to generate quantified formulas with any number of distinct bound variables. To do this it represents them as Prolog variables (the variable X in the clause (*)). Each instance of (*) gives a distinct version of X. None of them is ever instantiated. Under the annotation $sentence(\downarrow, \uparrow)$ the program is not simple. (End of example)

A program using variables as unique objects may be transformed into a simple one by exchanging these variables for atoms. This requires a procedure which generates unique atoms (see the procedure *gensym* [CM] p.149). Such a procedure must use extra-logical features of Prolog.

6b. Variable as a tag.

One of the meaningful values of a certain variable in a program is the variable uninstantiated. It may be used to find out that a certain action (which binds the variable) has not been performed. To check the value of the variable procedure *var* is used. This use of variables was found in SPOQUEL and INTERFACE.

6cde. The remaining cases of using variables as data can be treated as different ways of applying a programming technique of top-down construction of terms. The idea is to create first a non-ground term to be updated later by instantiating some of its variables. A final result may, or may not, be a ground term.

6c. Natural use of variables.

Two versions of something which can conveniently be called a natural use of variables as data were found. The first consists in using nonground terms to represent Prolog terms (including literals and clauses). This occurs in INTERFACE. The program contains a parser which reads and parses Prolog input. The result of parsing may then be asserted using *assert* or executed using *call*. So the most natural form of parser's output is Prolog terms itself.

The second version was found in WARPLAN and STORY. A term is used to represent a class of all its ground instances. Variables occurring in such a term may be instantiated later or may occur in a final result. This usage is very natural and quite convenient but in some cases it did not work and a ground representation for the same term classes is also used in WARPLAN.

6d. Open data structures.

An example is an open tree with variables as leaves. Such a tree grows by instantiating its leaves to new subtrees. It never becomes ground but each variable may be instantiated later. Open lists are used in INTERFACE and SPOQUEL.

6e. Variables as pointers.

This concerns the non-simple programs which do not belong to the previous classes. They use the technique of top-down building of terms but one variable can occur in two distinct terms. Then, changing one of the terms may change the other. One could imagine two terms sharing a variable as connected by a two-directional pointer. Instantiating the variable affects both terms. An example is a difference list which consists of two terms: an open list and a variable. Binding the variable to a list has as a side effect appending this list to the open list. Here is another example found in our benchmark.

Example (program serialise from WPB).

```
serialise(L,R) :-
```

```
            pairlists(L,R,A),
            arrange(A,T),
            numbered(T,1,N).
    pairlists([X|L],[Y|R],[pair(X,Y)|A]) :-
            pairlists(L,R,A).
    pairlists([],[],[]).
```

For a given ground list *Items*, the procedure *pairlists* produces a list of variables *SerialNos* and a (nonground) list *Pairs*. All the variables occurring in *SerialNos* occur also in *Pairs*. *SerialNos* is the final result of the computation. But before *serialise* succeeds, the list *Pairs* is processed by *arrange* and *numbered*. This results in instantiating all its variables to ground terms. It means that *SerialNos* is also made ground for it shared these variables.

Note that the declarative reading of the procedure is easy but this kind of discussion we have just made is necessary to convince ourselves that the procedure will work. (End of example)

5. CONCLUSIONS

In this paper, the notion of a conventional logic program was formalized as a class of simple logic programs. An experiment was performed to check whether actual programs belong to the class. Results seem to confirm the hypothesis that many logic programs resemble programs in conventional languages. About half of the analyzed programs turned out to be simple. Many other can be treated as simple programs with exception for their minor fragments. The programming techniques which led to "non-simpleness" were classified and discussed. A list of them is given in the previous section.

The annotations generated by the CHECKER for simple logic programs facilitate understanding of these programs. CHECKER can also be helpful in discovering bugs in programs intended to be simple.

As many actual programs are simple and so are substantial parts of the remaining ones, it is worthwhile to study techniques for efficient execution of simple logic programs. On the other hand, an implementation for simple programs only would be too restrictive. It would reject quite a lot of programs which programmers actually write. This suggests that an implementation using such special techniques should be able to accept any programs (using traditional methods for these program fragments for which the special ones are inapplicable). This suggestion seems to be valid also for the methods of [K] [B] [B1] as the restricted logic programming languages introduced there are very close to the class of simple logic programs (see also section 2). It should be mentioned, however, that a problem of separating non-simple parts of “otherwise simple programs” is not dealt with in this paper.

An interesting problem is an extension of the class of simple logic programs in order to include cases 1, 2, 6a and 6b (always failing clause, insignificant value, variables as unique objects, variable as a tag). They have a common property that when a nonground term is passed then its variables will never be instantiated. (By “a term is passed” we mean that it occurs in an input position at the moment of call or in an output position at the moment of success.) One may say that nonground data are treated in the same way as ground ones because their variables are never bound. It may be expected that implementation techniques for simple logic programs can also be used for programs of cases 1, 2, 6a and 6b.

ACKNOWLEDGEMENTS

Thanks are due to Jan Małuszyński for suggesting this area of research and for many valuable discussions. Some of the programs were obtained from Jan Komorowski. Feliks Kluźniak made interesting remarks on the first draft of this paper. Ivan Rankin helped to correct my English. This research has been partially supported by the National Swedish Board for Technical Development, projektnummer STUF 85-3166 and STU 86-3372.

REFERENCES

[AE] Apt, K.R. and van Emden, M.H., “Contributions to the Theory of Logic Programming”, J.ACM. 29, 841-862 (1982).

[B] Bruynooghe, M., “Compile time garbage collection”, IFIP TC2 Working Conference on Program Specification and Transformation, Bad Tölz, FRG, 1986, North Holland - in print

[B1] Bruynooghe, M., “Is logic programming real programming”, IFIP TC2 Working Conference on Program Specification and Transformation, Bad Tölz, FRG, 1986, North Holland - in print

[BPW] D.L.Bowen, L.Byrd, F.C.N.Pereira, L.M.Pereira and D.H.D.Warren, “Prolog-20 user’s manual”, 1984

[CM] Clocksin,W.F., Mellish,C.S., “Programming in Prolog”, Springer Verlag 1981

[DiM] Dembiński,P., Małuszyński,J., “AND-paralelism with intelligent backtracking for annotated logic programs”, 1985 IEEE Symposium on Logic Programming, Boston July 1985, IEEE Computer Society Press, 29-38

[DtM] Deransart,P. and Małuszyński,J., "Relating Logic Programs and Attribute Grammars", Journal of Logic Programming 3, No. 2 (1985) 119-158

[FH] Franzen,P., Hoffmann,B., "Automatic determination of data flow in extended affix grammars", 9th annual GI conference, Bonn 1979

[GKS] Grudzinski,W., Kluźniak,F., Szpakowicz,S., "SPOQUEL interpreter", Institute of Informatics, Warsaw University, 1982

[K] Kluźniak,F., "Type synthesis for ground Prolog", 4th International Conference on Logic Programming, Melbourne 1987

[KM] J. Komorowski and J. Małuszyński , "Logic programming and rapid prototyping", Research Report LITH-IDA-R-86-20, The Dept. of Computer and Information Science, Linköping University, 1986 (to appear in "Science of Computer Programming")

[KS] Kluźniak,F., Szpakowicz,S., "Prolog for programmers", Academic Press 1985

[M] Mellish,C.S., "Some global optimizations for a Prolog Compiler", J. Logic Programming 1985:1:43–66

[MK] Małuszyński, J. and Komorowski, J., "Unification-Free Execution of Logic Programs", 1985 IEEE Symposium on Logic Programming, Boston July 1985, IEEE Computer Society Press, 78–87

[R] Reddy, U.S., "Transformation of logic programs into functional programs", 1984 IEEE International Symposium on Logic Programming, IEEE Computer Society Press, 187–196

[W] Warren, D.H.D., "WARPLAN: a system for generating plans", DCL Memo 76, Department of Artificial Intelligence, University of Edinburgh Scotland, 1974.

[W1] Warren, D.H.D., "Implementing Prolog - compiling predicate logic programs", DAI, Research Report Nos 39 and 40, University of Edinburgh, 1977

APPENDIX 1. Simple logic programs can be run without occur check.

Executing simple logic programs without occur check is sound in the sense that no derivation can be successful in which two non-unifiable terms are "unified" by Prolog unification without occur check. While executing a simple logic program, omitting occur check may only delay a failure which should be otherwise caused by the check. The delay may sometimes lead to an infinite loop but all the answers obtained are correct with respect to the declarative semantics of logic programs.

The unification without occur check will be called p-unification. It produces infinite terms as results of p-unifying of term pairs which are not unifiable because of occur check. To prove soundness of executing simple logic programs without occur check we show that every result given by a computation using p-unification can also be produced using true unification. We manage to make the proof without a detailed definition of p-unification; only an assumption is needed that a finite ground term is not p-unifiable with an infinite term.

Theorem.

Let P be a simple logic program, p be an atomic formula with input positions ground and $C = q{:}{-}q_1, ..., q_m$ be a clause of P. If p and q are not unifiable or if p contains infinite terms then there exists no successful derivation of P using p-unification and starting from p and C.

Proof (By a "successful derivation" we mean an SLD-refutation in the terminology of [AE])

Assume that such a derivation exists. If the length of the derivation is 1 then $m = 0$, every variable in an output position of q occurs also in an input position. Thus every variable has to be unified with a ground term. If p contains infinite terms then p-unification fails (since an infinite term is not p-unifiable with a ground one). Hence contradiction, a derivation required by the theorem, of length 1 does not exist (for any p and C).

Now assume that $n > 1$ is the least number for which such a derivation exists, p is the related goal and $C = q{:}{-}q_1, ..., q_m$ is the clause. At the beginning of the derivation p-unification of p and q succeeds. Then there exists a variable occurring in C which does not occur in an input position of q and becomes bound to an infinite term by the p-unification of p and q. Then the first occurrence of such a variable in the right hand side of C, say in q_i, is in an output position of q_i. When q_i becomes a current goal (instantiated to, say, q_i') its input positions are ground and an infinite term occurs in an output position. The derivation beginning from q_i' is successful but shorter than n. Contradiction. This completes the inductive proof of the theorem.

APPENDIX 2. Mode analysis of simple logic programs.

During execution the input positions of a current goal are always ground so their mode [M] is ++. Mode analysis for output positions can be based on the following reasoning. Consider a clause C of a simple program and a literal occurring in its right hand side. Let us discuss instantiations of the variables of this literal when it becomes a current selected goal. All the variables for which there exists a defining occurrence preceding the literal are instantiated to ground terms. Among the remaining variables, those which do not occur in the clause head are unbound. This local analysis is able to produce full mode information about all variables except for those which

1. occur in the head but not in input positions *and*

2. have their first occurrence in the right hand side in the literal under consideration.

Using the mode information for clause variables it is trivial to find out modes for the output positions of the literal. These modes are valid for every instantiation of this literal as a selected goal. Now, all the literals with the same procedure name which occur in the right hand sides of the clauses (including the goal clause) should be taken into account. Combining their local modes, a (correct) mode declaration for the procedure can be generated.

The discussion above shows that it is possible to obtain a fair amount of mode information for simple logic programs using a one pass algorithm. Although abstract interpretation [M] can generate more precise mode declarations, it requires many passes over a program (transformed into a form of equations). The upper limit on the number of passes can be very high. So the approach discussed above may be a good compromise between efficiency of analysis and exactness of generated mode declarations.