LISTLOG - A PROLOG EXTENSION FOR LIST PROCESSING

Zsuzsa Farkas

Computer Research and Innovation Center - SZKI
H - 1015 Budapest, Donati utca 35-45, Hungary

Abstract

In this paper an alternative list representation for logic programs is introduced, based on so-called segment variables. These variables represent a whole sublist (segment) of a list, that is, when substituting such a variable by a list, not the list itself, but its elements are considered the elements of the original list. The notion of segment variables was first introduced in the LISP70 pattern matcher [1], and was suggested to be used in PROLOG by Marc Eisenstadt, as a step towards a more human man-machine interface for PROLOG. The original motivation for using these variables was to simplify the definition of some basic list processing predicates, mainly by avoiding recursion.

However, we have shown that this list representation has an even more important advantage: it brings the declarative and the procedural semantics of several list handling predicates nearer to each other, e.g. allowing a more complete set of solutions or avoiding some infinite loops.

LISTLOG is a PROLOG extension, handling these list expressions; it is implemented as a front-end to PROLOG, providing an extended matching algorithm.

1. List expressions with segment variables

A segment variable represents a whole segment (sublist) of a list. For example, in a PROLOG list expression

[a, X , b] (*)

substituting

X <-- [c , d]

the list will contain [c ,d] as the second element. Intuitively, however, one may want to have

[a , c , d , b]

as the result of such a substitution. To allow this, we introduce a new type of variable, called segment variable, handled in a special way: it can be substituted only by a list expression, and when such a variable is substituted by a list, this list becomes s sublist of the original list. For segment variables we will use a '^' prefix to distinguish them from normal variables. That is,

[a , ^ X, b]   will become   [a , c , d , b]

when the substitution X <-- [c,d] is applied (cf previous example (*)).

This is a quite natural extension of normal PROLOG lists: in the expression

[a , b | X]

X also represents a whole (final) segment of the list, but here we have the restriction that only the variable after the vertical bar is handled in this way. Our generalization simply means, that we allow such a variable not only at the last position. The above list can be rewritten in LISTLOG as

[a , b , ^ X]

representing the lists beginning with the elements a and b and continuing with any number of any elements. Similarly,

[a , ^ X , b]

can be used to represent the lists beginning with a, ending with b, and containing any number of any elements in between.

In LISTLOG we allow all the normal PROLOG expressions, only the list expressions are different:

The syntax of expressions:

<expression> ::= <list expression> | ...

<list expression> ::= [<list_elem>, ..., <list_elem>]

<list elem> ::= <segment variable> | <expression>

<segment variable> ::= ^ <variable>

2. Defining list handling predicates in LISTLOG vs PROLOG

The main motivation for introducing this list representation was to provide means for defining list handling predicates in a less algorithmic way as it is possible in PROLOG. Though these PROLOG definitions might be understandable to be read, it may cause difficulty for a naive user to formulate e.g. the "between" relation (see below). The main problem with these PROLOG definitions is the use of recursion, and a certain algorithmic approach. An other thing which is not easy to be accustomed to is the assymetry of the PROLOG lists: the first and the last elements of a list are handled in a completely different way.

However, when these problems do exist for the naive users, we are aware that this is not a central topic in logic programming and therefore it is not worthwhile to distract the users' attention from the main points such as declarative approach, backtracking, etc. Defining an alternative list representation and providing a suitable unifying algorithm, we achieve such an extension to PROLOG which might be used easier, though only in the special area of list processing.

In the following you can compare the definition of some basic list handling predicates:

in PROLOG

```
member(X,[X | L]).
member(X,[Y | L]) :-
      member(X,L).

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

first_elem([X|L],X).
last_elem([X],X).
last_elem([Y|L],X) :-
     last_elem(L,X).

sublist(SL,L):-
      append(PREV,SL,LL),
        append(LL,LATER,L).

between(X,Y,B,[X|L]):-
      until(L,Y,B) .
between(X,Y,B,[Z|L]) :-
      between(X,Y,B,L) .

until([Y|_],Y,[]) .
until([Z|L],Y,[Z|B]) :-
      until(L,Y,B) .

reverse([],[]).
reverse([A|L],RL):-
      reverse(L,LL),
        append(LL,[A],RL).

palindrome([]).
palindrome([A|L]) :-
      palidnrome(L1),
        append(L1,[A],L).
```

in LISTLOG

```
member(X,[^PREV, X , ^LATER]).

append(L1,L2,[^L1, ^L2]).

first_elem([X, ^L],X).
last_elem([^L, X], X).

sublist(SL, [^PREV, ^SL, ^LATER]).

between(X,Y,B,[^L1,X,^B,Y,^L2]).

reverse([],[]).
reverse([A,^L],[^LL,A]) :-
       reverse(L,LL).

palindrome([]).
palindrome([A , ^L, A] ):-
      palindrome(L).
```

3. Advantages of LISTLOG to PROLOG

a. Solutions in a more concise form

A difference between the PROLOG and LISTLOG list representation is that some object sets which in PROLOG may be described only by infinitely many expressions, in LISTLOG can be represented by a single expression. For example, those lists containing the constant '1' as element can be described in PROLOG by the following expressions:

```
[1 | T]
[X1, 1| T]
[X1, X2, 1 | T]
  ...
```

while in LISTLOG by

```
[ ^X , 1, ^T ]
```

As a consequence of this, some goals having an infinite sequence of solutions in PROLOG, will have only a single one in LISTLOG, of course, having the same meaning - representing the same set of objects.

The simplest example illustrating this difference is:

```
? member(1,L).
```

in PROLOG:

```
L = [1 | T]
  = [X1 , 1 | T]
  = [X1,X2,1 | T ]
   ....
```

in LISTLOG :

```
L = [^X , 1 , ^T]
```

b. Producing a more complete set of solutions

As it is well known, even in pure PROLOG there are differences between the declarative and procedural semantics. One of the aspects of this is that some of the logically valid solutions are not produced by the PROLOG execution mechanism. The following example illustrates a situation when in LISTLOG a more complete set of solutions is gained:

```
? member(1,L) and member(2,L).

 in PROLOG                          in LISTLOG

L = [1,2|T]                      L = [^X, 1 , ^Y, 2 , ^Z]
  = [1,X1,2 |T]                    = [^X, 2 , ^Y, 1 , ^Z]
  = [1,X1,X2,2|T]
  ....
```

The solution set produced in PROLOG is rather restricted: we can never find a list in which '2' preceeds '1', moreover, '1' is always stuck in the first position. In LISTLOG there are only two solutions, but they describe all the logically possible solutions. Note that this difference is a consequence of the property dealt with in the previous section, that is, of the possibility of describing a larger set of objects by an expression in LISTLOG than in PROLOG. The reason why PROLOG gives only these solutions is that the second subgoal has infinitely many solutions already for the first solution of the first subgoal, and the interpreter would return to the first subgoal only after exhausting all the possible solutions of the second subgoal. In LISTLOG the more concise form of the solutions makes possible to avoid this.

c. Avoiding infinite loops

A third problem with the PROLOG execution mechanism is that in some cases it produces an infinite loop instead of a negative answer, as in the following examples:

```
? member(1,L) and not member(1,L).

? next_to(X,Y,L) and not preceeds(X,Y,L).

? sublist([1],[2]).

? first_elem(L,1) and last_elem(L,2) and palindrom(L).
```

The structure of the infinite loop is that a first subgoal has infinitely many solutions, all of them refused by a second subgoal. These type of goals are answered with 'NO' in LISTLOG, again due to the more concise form of the solutions.

4. Unification in LISTLOG

As we have extended the notion of expressions, an extended unification algorithm must be provided as well. Before presenting such an algorithm, first revisit the general definitions for unification and then those properties different in PROLOG and LISTLOG. (A summary of general unification is found in [3]).

a. General notions of unification

instantiating an expression means substituting simultaneously some of its variables by other expressions and performing the possible simplifications. The expression produced by instantiating is called an instance of the original one.

simplifying an expression means replacing those subexpressions whose arguments become known by the value of the function at the given arguments

an expression E is the **unifier** of two expressions E1 and E2 if it is an instance of both of them, belonging to the same substitutions.

if U1 and U2 are unifiers of two expressions, then U1 is **more general** than U2 if U2 is an instance of U1.

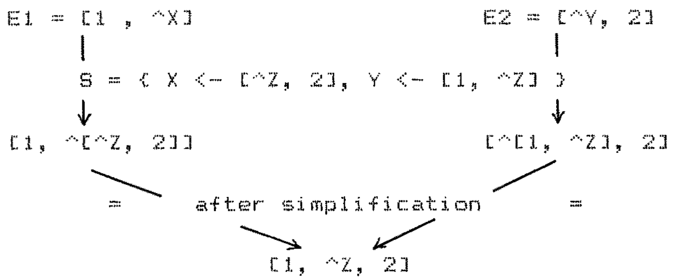
b. Simplification in LISTLOG

In the above definition only the notion of simplification depends on the given formal system: e.g. in PROLOG no simplification is needed since in Herbrand interpretations functions are defined having the function expressions themselves as values.

Simplification in LISTLOG means to flatten the elements of the list substitituted for a segment variable into the elements of the list containing this variable. E.g. the expression

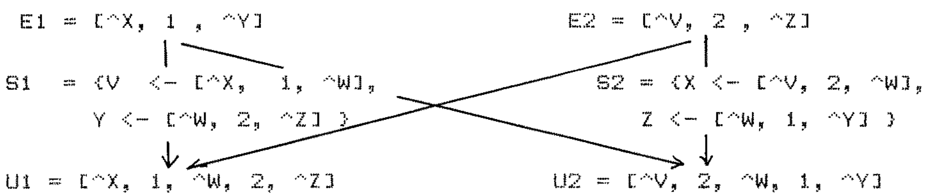
[a , X , ^ [c , Y], b] is simplified to [a , X , c , Y b]

The following expressions e.g. can be unified in LISTLOG :



c. Maximally general unifiers

As we know, in PROLOG there always exists a most general unifier for any two unifiable expressions, and the PROLOG unification algorithm is a deterministic procedure, giving this unifier. In LISTLOG this is not true: there may be expressions having not comparable unifiers, as it is illustrated by the following example:



that is, U1 and U2 are both unifiers of E1 and E2, and none of them is an instance of the other.

There may be infinitely many incomparable unifiers: these two expressions have the following unifiers:

E1 = [1, ^X]   E2 = [^X, 1]

U1 = []
U2 = [1]
U3 = [1,1]
...

It follows from the above that here we may have only maximally general unfiers instead of most general ones.

5. A unification algorithm

According to the above, our unification algorithm will be nondeterministic, producing each of the maximally general unifiers.

The algorithm presented here is a natural extension of the unfication used in PROLOG. The difference is that in PROLOG the only constructors are 'nil' and '.' , while in LISTLOG also the 'append' function is considered as constructor. (This corresponds to the list of form [^X,...]). We denote the list that results by appending the lists PRE and SUF together by

PRE .. SUF

The unfication algorithm is based on the following properties of lists:

(1) L1 = [] ==> L1 = L2 <==> L2 = []

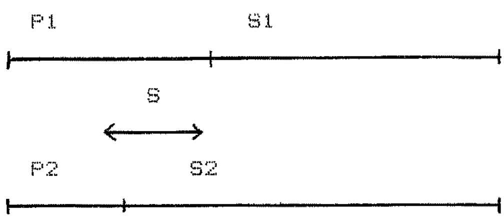
(2) H1 . T1 = H2 . T2 <==> H1 = H2 and T1 = T2

Properties (1) and (2) are used in PROLOG; further properties, (3) and (4), are added for LISTLOG unification:

(3) P1 .. S1 = P2 .. S2 <==> P1 = P2 and S1 = S2 or

P1 = P2 .. S and S2 = S .. S1 or

P2 = P1 .. S and S1 = S .. S2.



(4) P .. S = H . T <==> P = [] and S = H . T or

P = H . P1 and P1 .. S = T

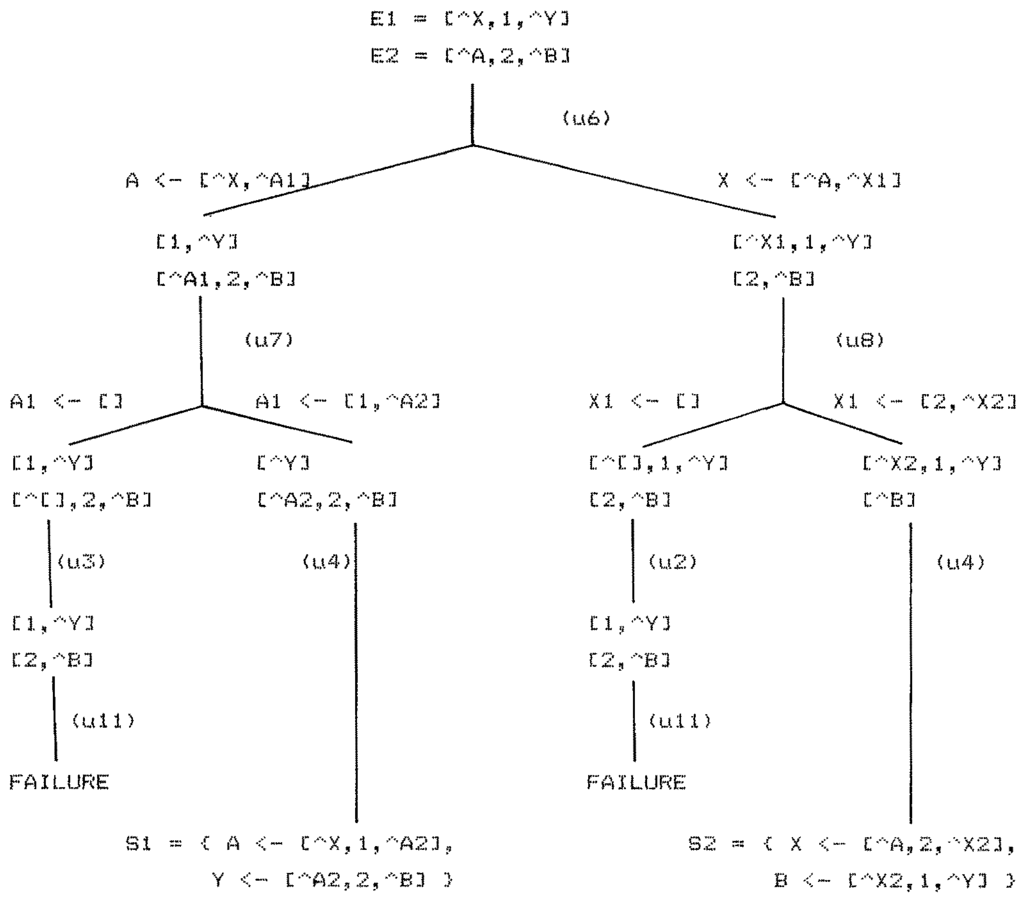
The unification defined in PROLOG

```
      operator(^, fx, 700).

      unify(X,Y) :-
        elem(X), !, X=Y .
      unify(X,Y) :-
        elem(Y), !, X=Y .
      unify(X,Y) :-
        is_list(X), !, unify_lists(X,Y);
        is_list(Y), !, unify_lists(X,Y) .
      unify(X,Y) :-
        decomp(X,[N|AL1]),comp([N|AL2],Y),unify_args(Al1,Al2).

      unify_args([],[]) .
      unify_args([A|L1],[B|L2]) :-
        unify(A,B), unify_args(L1,L2) .

(u1) unify_lists([],[]) .
(u2) unify_lists([S|T],L) :-
        bound_segment(S,SX), !, simplify(SX,S1), app(S1,T,L1),
          unify(L1,L) .
(u3) unify_lists(L,[S|T]) :-
        bound_segment(S,SY), !, simplify(SY,S1), app(S1,T,L1),
          unify(L,L1) .
(u4) unify_lists([S],L) :-
        unbound_segment(S,S1), !, S1=L .
(u5) unify_lists(L,[S]) :-
        unbound_segment(S,S1), !, S1=L .
(u6) unify_lists([S1|T1],[S2|T2]) :-
        unbound_segment(S1,X), unbound_segment(S2,Y), !,
            (X == Y, ! , unify(T1,T2);
             X=[^ Y,^ Z], unify([^ Z|T1],T2);
             Y=[^ X,^ Z], unify(T1,[^ Z|T2])) .
(u7) unify_lists([H|T1],[S|T2]) :-
        unbound_segment(S,X), !,
            (unify(X,[]), unify([H|T1],T2);
             unify(X,[H,^ Z]), unify(T1,[^ Z|T2])) .
(u8) unify_lists([S|T1],[H|T2]) :-
        unbound_segment(S,X), !,
            (unify(X,[]), unify(T1,[H|T2]);
             unify(X,[H,^ Z]), unify([^ Z|T1],T2)) .
```

```
(u9) unify_lists([S|T],[]) :-
       unbound_segment(S,X), !, unify(X,[]), unify(T,[]) .
(u10) unify_lists([],[S|T]) :-
       unbound_segment(S,X), !, unify(X,[]), unify(T,[]) .
(u11) unify_lists([H1|T1],[H2|T2]) :-
       unify(H1,H2), !, unify(T1,T2) .

      elem(X) :-
         X==[], !, fail;
         var(X), !;
         constant(X), !.

      is_list(L) :-
         var(X), ! , fail;
         L=[], !;
         L=[_|_] .

      bound_segment(S,X) :-
         var(S), !, fail;
         S=(^ X), (var(X), !, fail;
                   !) .

      unbound_segment(S,X) :-
          var(S), !, fail;
          S=(^ X), var(X).

      simplify(X,X) :-
          elem(X), !.
      simplify(L1,L2) :-
          is_list(L1), !, simplify_list(L1,L2) .
      simplify(X,Y) :-
          decomp(X,[N|AL1]), simplify_args(AL1,AL2),comp([N|AL2],Y).

      simplify_args([],[]) .
      simplify_args([A|L1],[B|L2]) :-
          simplify(A,B), simplify_args(L1,L2) .

      simplify_list([],[]) :-
          ! .
       simplify_list([E|L],[E1|L1]) :-
          unbound_segment(E,_), !, E1=E, simplify_list(L,L1).
```

```
simplify_list([E|L],EL) :-
   bound_segment(E,X), !, simplify(X,X1),
      simplify_list(L,L1), app(X1,L1,EL).
 simplify_list([E1|L1],[E2|L2]) :-
   simplify(E1,E2), simplify_list(L1,L2) .

 app([],L,L) .
 app([A|L1],L2,[A|L]) :-
   app(L1,L2,L) .
```

An example unification



U1 = [^X,1,^A2,2,^B]

U2 = [^A,2,^X2,1,^Y]

6. Transforming LISTLOG statements into PROLOG

As the only difference between PROLOG and LISTLOG lies in the data structures they handle, the only necessary transformation is to use our unification algorithm instead of the normal PROLOG one. This is done in such a way that the expressions occurring in the head of a statement are substituted by (new) variables, and the above unification algorithm is called explicitly before executing the body of the statement. E.g.

```
member(X,[^P,X,^S]).         =====>    member(X,Y) :-
                                          unify(Y,[^P,X,^S]).

append(L1,L2,[^L1,^L2]).  =====>    append(L1,L2,L) :-
                                          unify(L,[^L1,^L2]).
```

An additional problem is how to call the PROLOG built-in predicates such as 'write'. In the above execution scheme simplification is performed during the unification when entering a definition. However, this causes a problem in the case of built-in predicates, because here we cannot apply this transformation. This is solved in our system in such a way, that a predicate

```
call(CONDITION)
```

is introduced which provides an interface for PROLOG predicates: before evaluating the given condition it simplifies its arguments. E.g.

```
append([1,2,3],[4,5],L),call(write(L)).
```

gives the expected output [1,2,3,4,5], the simplified form of [^[1,2,4],^[4,5]] produced by the "append" definition.

7. Efficiency questions

In the case of the above PROLOG unification understandability had a higher priority than efficiency. However, it is worth mentioning that there are cases when even this implementation increases efficiency, compared to PROLOG. For example, a question of the form

```
? reverse([1,2,3,4,5,6,...,n] , []).
```

in PROLOG can be answered negatively only by actually reversing the list, while in LISTLOG this answer is produced by a single

```
unify([^X, 1], [] )
```

unification step, independently of the length of the list to be reversed.

8. Conclusions, further plans

We have shown the advantages of an alternative list representation for logic programming. LISTLOG, the resulted PROLOG extension is implemented in PROLOG at the moment, but we are considering a more efficient direct implementation for it. Further directions are to try to generalize this method from lists to other PROLOG structures, or remaining in the list processing area, to refine further the above list representation [4].

References

[1] L.G.Tesler - H.J.Enea - D.S.Smith:
The Lisp70 pattern matching system.

[2] M.Eisenstadt:
An improved man-machine interface for PROLOG.
Imperial College - Open University joint project, 1984.

[3] J.H.Stickman:
Universal unification.
7th International Conference on Automated Deduction, LNCS 170, Springer Verlag,1984.

[4] Zs. Farkas:
Length restricted segment variables in PROLOG.
(in preparation)