Self-correcting Generalization

Stephen B. Whitehill

Technical Report 149

June 1980

ABSTRACT. A system is described which creates and generalizes rules from examples. The system can recover from an initially misleading input sequence by keeping evidence which supports (or doesn't support) a given generalization. By undoing over-generalizations, the system maintains a minimal set of rules for a given set of inputs.

0 Introduction

Many programs have been written which generalize examples into rules. Soloway's program[5] makes generalizations about the rules of baseball from examples. Hayes-Roth[3] and Winston[7] generalize common properties in structural descriptions. Vere[6] has formalized generalization for several applications.

If a program maintains a current hypothesis about the rule set as it sees new examples it is said to generalize incrementally. Any program that incrementally forms generalizations is sensitive to the order in which examples are presented. If exceptional examples are encountered first, the program may over-generalize. If the program is to recover and undo the over-generalization it must have a certain amount of knowledge into why the over-generalization was made.

The system to be described here has this type of self-knowledge. By associating positive and negative evidence with each generalization, the system is able to reorganize its rules to recover from over-generalizations. Even if it is initially misled by an unusual sequence of inputs it still discovers the most reasonable set of rules.

This paper has 6 sections. The first section describes the problem domain and Concept Description Language. The next two sections describe the relationships that can exist

between rules and how the system uses these to organize (and reorganize) its rule system. Then, the concept of a minimal rule set is discussed. Then, a detailed example of the program learning the rules for French adjectives is presented. Finally, a detailed example of the system reorganizing its rules to recover from misleading input is shown.

1 The Problem Domain

The problem domain chosen for the system is learning language morphology from examples. For example, from the words "jumped", "walked" and "kissed" we can deduce the rule that the English past tense is formed by adding "ed".

The Concept Description Language (CDL) consists of a set of rules. Each rule is a production consisting of a pair of character strings. When the left-hand side is matched the right-hand side is the result. The left-hand string may optionally start with a '*' which will match zero or more characters. In this case the right hand character string may start with a '*' and the value the star on the left hand side which was matched is substituted for the star on the right hand side. For example the production for the example above looks like: *->*ED. This *->*ED rule does not always work. From "baked", "related" and "hoped" we see that for words ending in "e" we need only to add an "d". This rule is written as *E->*ED.

Note that we also could have deduced *->*D from the 3 examples above. Thus, for a given set of examples, a number of generalizations may be made. Fortunately the Maximal Common Generalization (MCG) as defined by Vere[6] is unique for this CDL. This will be proved in a later report focusing on the formal aspects of the generalizaton algorithm for this CDL. The MCG is the generalization that captures all the common features of the examples. Thus for "churches" and "matches" the MCG would be *CH->*CHES rather than *H->*HES or *->*ES. It is important to note that the MCG may not necessarily be unique for other CDL's.

2 Relationships Between Rules

Rule Containment. Given rules P1 and P2, let S1 be the set of strings which match the left-hand side of P1 and S2 the set of strings which match left-hand side of P2. If S1 is a subset of S2 then we say that P1 is contained by P2.

There are three possible ways two rules may be related by rule containment. One rule may contain another, it may be contained by the other, or they may not be related by rule containment. In brief, rule containment is a partial ordering of the rules. If two rules have the same left-hand side they contain each other.

The Is-a-generalization-of Operator. Given rules P1 and P2, let S1 be the set of strings which match the left-hand side of P1 and S2 the set which match the

left-hand side of P2. If P1 contains P2 and if P1 and P2 produce the same result for every element in S2 then P1 is a generalization of P2. Vere[6] calls this operator >. This is also a partial ordering of the rules.

Note the distinction between the containment operator and the is-a-generalization-of operator. Basically, containment deals with the left-hand side of a rule. Is-a-generalization-of deals with both sides. This example will clarify the difference:

*->*S contains *K->*KS
*->*S is a generalization of *K->*KS
*->*S contains *CH->*CHES
*->*S and *CH->*CHES are unrelated by generalization

By definition, if a rule is a generalization of another it contains the other. The converse is not necessarily true.

If P1 is a generalization of P2 and P1 is a generalization of P3 then P1 is a common generalization of P2 and P3. P1 is a maximal common generalization[6] if P1 is a generalization of no other common generalization. Roughly, the maximal common generalization is the one which captures all common features of the rules being generalized. For example, given WALK->WALKED and TALK->TALKED, possible generalizations are: *->*ED, *K->*KED, *LK->*LKED and *ALK->*ALKED. The last one, *ALK->*ALKED, is maximal.

3 Organization of Rules

The rules and their evidence are organized in a tree structure. At the top level the rules are organized as a rule list. A rule list is a list of rules partially ordered by the containment operator. No rule may be contained by a rule which precedes it in the list. Associated with most rules is some evidence which is itself in the form of another rule list. The only rules without evidence are the example pairs whose evidence is fact. These correspond to terminal nodes in the evidence tree.

If two adjacent rules can be generalized, the result R1 may contain a rule R2 that follows it. This would violate the partial ordering of the rules by the containment operator. To avoid this, the generalization R1 is made but is marked as being blocked by R2. Blocked rules are ignored by the algorithm used to find rules for a given input. Blocked rules are used only for rule reorganization. If R1 blocks R2 then evidence for R1 is negative evidence for R2.

The positive evidence consists of those rules which were generalized into the current generalization. Negative evidence for a generalization G is all the evidence of generalizations that are blocked by G. Thus when *->*ES blocks (*N->*NS + *K->*KS ==> *->*S) that is negative evidence for *->*ES. The evidence described here is much like evidence used to produce explanations in STAMMER[1] or to maintain beliefs in Doyle's system[2]. In our rule

system the evidence is used for reorganization, but it could be used for explanation or belief maintainence as well.

In LISP, the rule system is a list of atoms. On the property RULE of each atom in the list is the representation of a rule. For example, the rule *->*S would be represented in LISP as: ((*)(* S)). On the EVID property of an atom, if it exists, is a list of atoms which support that generalization. These atoms may have properties themselves. Thus for the input pairs:

book -> books, back -> backs and car -> cars

The environment would look like:

```
RULES = (G2)
GET(G2,'RULE) = * -> *S
GET(G2,'EVID) = (I3 G1)
GET(G1,'RULE) = *K -> *KS
GET(G1,'EVID) = (I2 I1)
GET(I1,'RULE) = BOOK -> BOOKS
GET(I2,'RULE) = BACK -> BACKS
GET(I3,'RULE) = CAR -> CARS
```

The rule pretty-printer in the program prints this as:

```
*->*S          CAR->CARS
               *K->*KS          BACK->BACKS
                                BOOK->BOOKS
```

Rule Application and Conflict Resolution. When the rule interpreter produces a response, it is as if it finds all rules which match the given input and then uses the one which is contained by all the others (informally, the one with the least general left-hand side). In reality, the rules and rule interpreter are organized so that the first rule that matches is the desired one. This technique uses the fact that all rules matching the input form a totally

ordered set. This is not true in general, but is for our CDL.

The algorithm used by the production system to find the most specific applicable rule recursively searchs the rules and evidence. When a blocked production is seen, its evidence (a more specific list of rules) is used instead. This process can be viewed as an in-order tree-traversal treating blocked rules as internal nodes and unblocked rules as terminal nodes. Here is the rule finding algorithm:

```
findrule(input, rulelist);
    begin new found;
    found _ nil;
    for rule in rulelist do
        if blocked(rule) then
            found _ findrule(input, evidence(rule))
        else
            found _ match(input, rule)
    until found;
    return found
    end;
```

The function match(input, rule) returns the substituted right of rule if the left side matches input. for example:

```
match(book, book->books) = books
match(car,  *->*s)       = cars
match(house, *ch->*ches) = NIL
```

Inserting New Rules. If a rule has produced the correct result, the new example pair is inserted into the evidence list for the rule. If the rule system has not produced the correct result the rule is inserted in the main rule list before the first rule with which it will generalize. If it will not generalize with any rule, it is inserted before the first rule that contains it. The same rule insertion algorithm is used to insert new rules or evidence. This means that generalizations take place in an evidence list in the same way that they do in the main rule list. Here is the algorithm used by the system to insert rules:

```
rinsert(r, rl);  % return a new rule list %
  if null rl then list(r)
  else if rl[1] contains r then
          if blocked(rl[1]) then
             begin
             evidence(rl[1])
                 _ rinsert(r, evidence(rl[1]));
             rl _ reorganize(rl);
               % if needed. see below.
             return rl;
             end
          else r cons rl
  else if not null mcg(r,rl[1]) then
         begin
         genrule _ mcg(r,rl[1]);
         evidence(genrule) _ list(r, rl[1]);
         b _ overgen(genrule,cdr rl);
           % b is rule blocking s or NIL %
         if not null b then
              blocked(genrule) _ b;
         return genrule cons cdr rl;
         end
  else rl[1] cons rinsert(r, cdr rl);    % recurse %
```

System Reorganization. Each blocked generalization has knowledge about which generalization is blocking it. Whenever evidence for a blocked generalization G1 is entered into the rule structure, we check to see if there is now more evidence for G1 than for the blocking generalization G2. If so, G2 is moved to the position in the rule list immediately preceding G1, if G2 is now blocked by G1 it is so marked, and G1 is no longer marked as being blocked.

There are several choices on how to compare positive and negative evidence. The one chosen is to count how much direct evidence there is for a rule. Direct evidence is that evidence found in the top level rule list in the evidence tree. Another method which was rejected for this application is to count the total number of pieces of evidence in the evidence tree. The first method was chosen because *CH->*CHES and *X->*XES are exceptions to *->*S (rather than *A->*AS, *B->*BS, *C->*CS, etc. being exceptions to *->*ES) because there is more direct evidence for *->*S (rules like *A->*AS) than for *->*ES. Even if half the words in English used *CH->*CHES this would still be an exception to *->*S.

It should be noted that the above algorithms are independent of the concept description language. The input could be structural descriptions or mathematical equations rather than language morphology. The only restriction is that the MCG be unique for the concept description language.

4 The Minimal Rule Set

A rule set is minimal if it explains all inputs which have been seen and there exists no rule set explaining the input which has fewer rules. Clearly, the minimal rule set is not unique. For example, consider the English past tense for: "find", "mind", "bind", "jump", "walk" and "match". One rule set would be:

```
MIND->MINDED
*IND->*OUND
*->*ED
```

Another would be:

```
FIND->FOUND
BIND->BOUND
*->*ED
```

Since these rule sets are both minimal, neither seems clearly better than the other.

The system appears to always produce a minimal rule set. We assert this without proof at the moment. A proof by induction of this claim is being studied. The following examples will show that minimal sets are produced.

5 Generalizing Rules for French Adjectives

The following example shows the system learning the rules for pluralizing French adjectives. The system was found to produce the same rules for pluralizing French adjectives as those found in a French-English dictionary.

Morphemic rules usually consist of one or more regular forms and many exceptions. In French, which has a much more complicated morphology than English, exceptions can be generalized. In this case there can be exceptions to exceptions. The following passage, taken from a French-English dictionary, describes the rules for pluralizing masculine French adjectives (which must agree in gender and number with the noun they modify).

> "the plural is generally formed by adding 's' to the singular form. Adjectives ending in 's' or 'x' do not change in the plural. Adjectives in 'al' form their plural in 'aux'. Ex: principal, principaux. But: bancal, glacial, natal, naval form their plural in 'als'"

Notice that there is a general rule: *->*S, some exceptions; *S->*S, *X->*X and *AL->*AUX, and exceptions to the *AL->*AUX exception.

The English speaking student learning French quickly learns the basic rule for pluralizing adjectives; * -> *s. With this rule he has no trouble with NAVAL->NAVALS. However as he studies further, he learns that NATIONAL->NATIONAUX, PRINCIPAL->PRINCIPAUX, ORAL->ORAUX, etc. This leads to the (correct) generalization *AL->*AUX. At this point NAVAL, which was regular, becomes irregular. The student will produce the incorrect form NAVAUX. He must learn that this is an exception to an exception. His final rule base is:

```
NAVAL->NAVALS
*AL->*AUX
*->*S
```

Which is much more efficient than:

```
NATIONAL->NATIONAUX, ORAL->ORAUX, MAL->MAUX ...
*->*S
```

It appears that language learners develop the same notion of a minimal rule set that the system does. The number of rules is what is minimized (as opposed to the length of the rule set or the complexity of the rules).

Here is an example of the program learning French adjectives. The system is written in MLISP and runs on UCI LISP. The following example was annotated from a DSKLOG of the system in operation. What the system typed is in upper case. The rules and evidence are printed as a tree. The evidence for a node is indented from the parent node in the printout.

```
Input word? facile
What is response? faciles
 FACILE->FACILES

Input word? grand
What is response? grands
 *->*S              GRAND->GRANDS
                    FACILE->FACILES

Input word? gentil
Is response GENTILS? y
 *->*S              GENTIL->GENTILS
                    GRAND->GRANDS
                    FACILE->FACILES
```

At this point the basic rule has been learned.

```
Input word? naval
Is response NAVALS? y
 *->*S              NAVAL->NAVALS
                    GENTIL->GENTILS
                    GRAND->GRANDS
                    FACILE->FACILES
```

It gets this right. It doesn't know this is an exception to an exception.

```
Input word? gross
Is response GROSSS? n
What is response? gross
 GROSS->GROSS
 *->*S              NAVAL->NAVALS
                    GENTIL->GENTILS
                    GRAND->GRANDS
                    FACILE->FACILES

Input word? principal
Is response PRINCIPALS? n
What is response? principaux
 GROSS->GROSS
 PRINCIPAL->PRINCIPAUX
 *->*S              NAVAL->NAVALS
                    GENTIL->GENTILS
                    GRAND->GRANDS
                    FACILE->FACILES
```

```
Input word? bas
Is response BASS? n
What is response? bas
 *S->*S              BAS->BAS
                     GROSS->GROSS
 PRINCIPAL->PRINCIPAUX
 *->*S               NAVAL->NAVALS
                     GENTIL->GENTILS
                     GRAND->GRANDS
                     FACILE->FACILES

Input word?oral
Is response ORALS? n
What is response? oraux
 *S->*S              BAS->BAS
                     GROSS->GROSS
 *AL->*AUX           ORAL->ORAUX
                     PRINCIPAL->PRINCIPAUX
 *->*S               NAVAL->NAVALS
                     GENTIL->GENTILS
                     GRAND->GRANDS
                     FACILE->FACILES
```

The system has learned new general exceptions.

```
Input word? naval
Is response NAVAUX? n
What is response? navals
 *S->*S              BAS->BAS
                     GROSS->GROSS
 NAVAL->NAVALS
 *AL->*AUX           ORAL->ORAUX
                     PRINCIPAL->PRINCIPAUX
 *->*S               NAVAL->NAVALS
                     GENTIL->GENTILS
                     GRAND->GRANDS
                     FACILE->FACILES
```

The system now knows that NAVAL is an exception to the *AL->*AUX exception.

6 Undoing Generalizations - an Example

The following is a DSKLOG of the system recovering from initially misleading inputs.

```
Input word? church
What is response? churches
Input word? match
What is response? matches
Input word? bunch
Is response BUNCHES? y
Input word? bus
What is response? buses
RULES:
 *->*ES          BUS->BUSES
                 *CH->*CHES      MATCH->MATCHES
                                 CHURCH->CHURCHES
```

At this point we have over-generalized. We will find this out later. The only rule seen by the rule interpreter is *->*ES. BUS->BUSES and *CH->*CHES are evidence for *->*ES. MATCH->MATCHES and CHURCH->CHURCHES are evidence for the *CH->*CHES rule (which is itself evidence).

```
Input word? book
Is response BOOKES? n
What is response? books
Input word? rack
Is response RACKES? n
What is response? racks
RULES:
 *K->*KS         RACK->RACKS
                 BOOK->BOOKS
 *->*ES          BUS->BUSES
                 *CH->*CHES      MATCH->MATCHES
                                 CHURCH->CHURCHES
```

What should be regular cases are treated as exceptions.

```
Input word? car
Is response CARES? n
What is response? cars
RULES:
 (*->*S)          CAR->CARS
                  *K->*KS          RACK->RACKS
                                   BOOK->BOOKS
 *->*ES           BUS->BUSES
                  *CH->*CHES       MATCH->MATCHES
                                   CHURCH->CHURCHES
```

At this point we want to make the generalization *->*S but this generalization is blocked by *->*ES. We make the generalization but mark it as blocked. The parenthesis indicated that the rule is blocked. The productions seen by the production system are:

```
CAR->CARS
*K->*KS
*->*ES
```

The blockage of *->*S is negative evidence for *->*ES. The system should detect that the rules are in the wrong order when there is more evidence for *->*S (and hence against *->*ES) than there is for *->*ES.

```
Input word? bar
Is response BARES? n
What is response? bars
RULES:
 (*->*S)          *AR->*ARS        BAR->BARS
                                   CAR->CARS
                  *K->*KS          RACK->RACKS
                                   BOOK->BOOKS
 *->*ES           BUS->BUSES
                  *CH->*CHES       MATCH->MATCHES
                                   CHURCH->CHURCHES
```

At this point there is just as much negative evidence as positive (looking down one level in the evidence tree).

```
Input word? bat
Is response BATES? n
What is response? bats
RULES:
 (*->*ES)          BUS->BUSES
                   *CH->*CHES        MATCH->MATCHES
                                     CHURCH->CHURCHES
 *->*S             *AR->*ARS         BAR->BARS
                                     CAR->CARS
                   *K->*KS           RACK->RACKS
                                     BOOK->BOOKS
                   BAT->BATS
```

This addition negative evidence for *->*ES has caused a reordering of the rules. *->*ES is now blocked by *->*S (as it should be).

```
Input word? house
Is response HOUSES? y
Input word? bunch
Is response BUNCHES? y
```

The system now has a properly ordered rule set and can handle both regular and irregular cases.

7 Conclusions

By giving a generalization program some self-knowledge it can recover from initially misleading input sequences. This introspection can be achieved by associating positive and negative evidence with generalizations. The system described here discovers the most reasonable set of morphological rules (minimal in terms of number of rules) for a given language construct. This set, which is the same set found in a dictionary, is found regardless of the input sequence. The choice of language morphology as the problem domain was arbitrary. Any domain with a concept description language whose maximal common generalization is unique would serve just as well. Further work is needed for concept description languages whose maximal common generalization is not necessarily unique. Any incremental generalization program could improve its ability to recover from misleading input by applying the techniques described.

8 References

[1] Bechtel, R., Morris, P. and Kibler, D., "Incremental Deduction in a Real-time Environment", Canadian Society for the Computational Studies of Intelligence (May 1980).

[2] Doyle, J., "A Glimpse of Truth Maintenance", Proceedings of the Sixth International Joint Conference on Artificial Intelligence (August 1979), 232-237.

[3] Hayes-Roth, F. and McDermott, J., "Knowledge Acquisition from Structural Descriptions", Department of Computer Science, Carnegie-Mellon University (1976).

[4] Smith, R. G.,Mitchell, T. M., Chestek, R. A. and Buchanan, B. G. "A Model for Learning Systems", Proceedings of the Fifth International Joint Conference on Artificial Intelligence (1977), 338-343.

[5] Soloway, E. M. and Riseman, E. M., "Levels of Pattern Description in Learning", Proceedings of the Fifth International Joint Conference on Artificial Intelligence (1977), 801-811.

[6] Vere, S., "Induction of Relational Productions in the Presence of Background Information", Proceedings of the Fifth International Joint Conference on Artificial Intelligence (1977), 349-355.

[7] Winston, P., "Learning Structural Descriptions From Examples", MIT-AI Technical Report 231, (1970).

Appendix I - code for learning system

```
EXPR TOP();
    BEGIN NEW X;
    PROMPT(63);
L;  MSG(T,"INPUT WORD");
    X _ CAR LINEREAD();
    IF X = 'R THEN PRINTRULES(RULES,2)
    ELSE IF X = 'N THEN RULES _ NIL
    ELSE IF NUMBERP(X) THEN GABBY _ (X=1)
    ELSE IF CONSP X THEN PRINT(EVAL(X)) ALSO MSG(T)
    ELSE LEARN(EXPLODE X);
    GO L;
    END;

MACRO STORE(L);
    %     RULE(A) _ B; compiles into:                %
    %     (STORE (RULE A) B) which this macro makes: %
    %     (PUT A B 'RULE)                            %
    <'PUT, L[2,2], L[3], <'QUOTE, L[2,1]>>;

EXPR RULE(A);     GET(A,'RULE);

EXPR EVIDENCE(A); GET(A,'EVIDENCE);

EXPR BLOCKED(A);  GET(A,'BLOCKED);

EXPR PRINTRULES(RL,SP);          % pretty-print rules %
    FOR NEW R IN RL DO:
        BEGIN NEW E;
        PRINTR((RULE R),SP,(BLOCKED R));
        E _ (EVIDENCE R);
        IF E THEN PRINTRULES(E,SP+16)
        ELSE MSG(T);
        END;

EXPR PRINTR(R,SP,BLFLAG);     % pretty-print auxilary %
    BEGIN
    MSG(T(SP));
    IF BLFLAG THEN MSG("(");
    MSG(READLIST(R[1]),"->",READLIST(R[2]));
    IF BLFLAG THEN MSG(")");
    END;

EXPR YESNO();                 % return T if YES input %
    BEGIN NEW Z;
L;  Z _ CAR LINEREAD();
    Z _ EXPLODE Z;
    IF Z[1]='Y THEN
        RETURN(T);
    IF Z[1]='N THEN
        RETURN(NIL);
    GO L;
    END;
```

```
EXPR LEARN(X);              % main learning algorithm %
    BEGIN NEW GLOBALR, FOUND;
    FOUND _ FINDRULE(X,RULES);
    IF NOT FOUND THEN
        RULES
          RINSERT(NEWRULE(X,READRESPONSE()),RULES)
    ELSE
        BEGIN
        MSG("IS RESPONSE ",READLIST FOUND);
        IF YESNO() THEN
            ADDEVID(GLOBALR,NEWRULE(X,FOUND))
        ELSE
          RULES
            RINSERT(NEWRULE(X,READRESPONSE()),RULES);
        END;
    IF GABBY THEN PRINTRULES(RULES,2);
    END;

EXPR FINDRULE(INPUT,RULELIST);
    BEGIN NEW R,FOUND;
    FOR R IN RULELIST DO:
        IF (BLOCKED R) THEN
            FOUND _ FINDRULE(INPUT,(EVIDENCE R))
        ELSE IF FOUND _ MATCH((RULE R),INPUT) THEN
            GLOBALR _ R
    UNTIL FOUND;
    RETURN FOUND;
    END;

EXPR ADDEVID(A,EV);      % adds evidence at the
                           deepest possible level %
    IF MATCH((RULE CAR(EVIDENCE A)),
             CAR(RULE EV)) THEN
        ADDEVID(CAR(EVIDENCE A),EV)
    ELSE (EVIDENCE A) _ EV CONS (EVIDENCE A);

EXPR READRESPONSE();
    BEGIN
    MSG("WHAT IS RESPONSE");
    RETURN EXPLODE CAR LINEREAD();
    END;

EXPR NEWRULE(INPUT,RESPONSE);
    BEGIN
    R _ GENSYM();            % creat new symbol %
    (RULE R) _ <INPUT,RESPONSE>;
    RETURN R;
    END;
```

```
EXPR RINSERT (R, RL, PARENT);
% insert rule R into rulelist RL. returns copy %
    IF NULL RL THEN <R>
    ELSE IF LG (CAR(RULE R), CAR(RULE RL[1])) THEN
           IF (BLOCKED RL[1]) THEN
           % if rule is block insert in evid. %
              (EVIDENCE RL[1])
                RINSERT(R, (EVIDENCE RL[1]), RL[1])
              ALSO CHECKSWAP(RL[1])
              ALSO RL
           ELSE R CONS RL
    ELSE IF (X _ MCG ((RULE R), (RULE RL[1])))
             AND X NEQUAL (RULE PARENT) THEN
           BEGIN NEW S,B;
           S _ NEWSYM G;
           (RULE S) _ X;
           (EVIDENCE S) _ <R, RL[1]>;
           IF B _ OVERGEN(X,CDR RL) THEN
                (BLOCKED S) _ B;
           RETURN S CONS CDR RL;
           END
  ELSE RL[1] CONS RINSERT (R, CDR RL, PARENT);

EXPR OVERGEN(P,L);
    IF NULL L THEN NIL
    ELSE IF ~LGORNR(CAR P, CAR (RULE L[1])) THEN
        L[1]
    ELSE OVERGEN(P,CDR L);

EXPR LGORNR (X, Y);   % LG or No Relation %
    X NEQUAL Y & (LG (X, Y) ! ~LG (Y, X));
```

```
EXPR CHECKSWAP(R);
    BEGIN NEW B;
    B _ (BLOCKED R);
    IF EVIDLEN(R) GT EVIDLEN(B) THEN
        SWAP(R);
    END;

EXPR EVIDLEN(R);
    BEGIN NEW E,N;
    E _ (EVIDENCE R);
    N _ 0;
    FOR NEW I IN E DO:
        IF (BLOCKED I) THEN
            N _ N + EVIDLEN(I)
        ELSE N _ N + 1;
    RETURN N;
   END;

EXPR SWAP(R);          % R is blocked rule %
    BEGIN NEW B;       % B is blocking rule %
    B _ (BLOCKED R);
    (BLOCKED R) _ NIL;
    IF ~LGORNR(CAR(RULE B), CAR(RULE R)) THEN
        (BLOCKED B) _ R;
    RL _ SWAPAUX(RL,R,B);  % RL special to RINSERT %
    END;

EXPR SWAPAUX(RLS,R,B);
% SWAPAUX((X R Y B),R,B) -> (X B R Y) %
    IF NULL RLS THEN NIL
    ELSE IF RLS[1]=R THEN
        <B,R> @ SWAPAUX(CDR RLS,R,B)
    ELSE IF RLS[1]=B THEN
        SWAPAUX(CDR RLS,R,B)
    ELSE RLS[1] CONS SWAPAUX(CDR RLS,R,B);
```

```
IMMEDIATE LETTER STAR;

EXPR MATCH(R,D);
    IF MTCH(R[1],D) THEN
        SUBS(R[2]);

EXPR MTCH(P,D);
% match P and D. P may contain a '*' %
    IF P[1] NEQ '* THEN P=D
    ELSE IF NULL CDR P THEN
        STAR_D ALSO T
    ELSE STAR_NIL
         ALSO MTCHAUX(P,D);

EXPR MTCHAUX(P,D);  % aux function for recursion %
    IF NULL D THEN NIL
    ELSE IF CDR(P) = D THEN T
    ELSE STAR _ STAR @ <D[1]>
         ALSO MTCHAUX(P, CDR D);

EXPR SUBS(P);              % used to creat result %
    IF P[1] NEA '* THEN P
    ELSE STAR @ CDR P;

EXPR LG(P1,P2); % Less general than %
% P1 = '(* T E D), P2 = (* E D); RETURN T %
    IF P1 = P2 THEN NIL
    ELSE IF NULL P2 THEN T
    ELSE IF P1[1]='* THEN
        MTCH(P2,CDR P1)
    ELSE MTCH(P2,P1);

EXPR MCG(PAIR1,PAIR2);
% find the least general common generalization %
    BEGIN NEW X1,X2,X3,X4;
    X1_PAIR1[1];
    X2_PAIR1[2];
    X3_PAIR2[1];
    X4_PAIR2[2];
    WHILE LENGTH(X1) GT LENGTH(X3) DO X1_CDR X1;
    WHILE LENGTH(X3) GT LENGTH(X1) DO X3_CDR X3;
    WHILE LENGTH(X2) GT LENGTH(X4) DO X2_CDR X2;
    WHILE LENGTH(X4) GT LENGTH(X2) DO X4_CDR X4;
    WHILE X1 NEQUAL X3 & X2 NEQUAL X4
          & X1[1] = X2[1] & X3[1] = X4[1] DO:
        BEGIN
        X1_CDR X1;
        X2_CDR X2;
        X3_CDR X3;
        X4_CDR X4
        END;
    IF X2 NEQUAL X4 ¦ X1 NEQUAL X3 THEN RETURN NIL
    ELSE RETURN < '(*) @ X1,'(*) @ X2>
    END;
```