HOW TO INVERT A NATURAL LANGUAGE PARSER INTO AN EFFICIENT GENERATOR: AN ALGORITHM FOR LOGIC GRAMMARS

Tomek Strzalkowski
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

ABSTRACT

The use of a single grammar in natural language parsing and generation is most desirable for variety of reasons including efficiency, perspicuity, integrity, robustness, and a certain amount of elegance. In this paper we present an algorithm for automated inversion of a PROLOG-coded unification parser into an efficient unification generator, using the collections of minimal sets of essential arguments (*MSEA*) for predicates. The algorithm is also applicable to more abstract systems for writing logic grammars, such as DCG.

INTRODUCTION

In this paper we describe the results obtained from the experiment with reversing a PROLOG parser for a substantial subset of English into an efficient generator. The starting point of the experiment was a string parser for English (Grishman, 1986), which is used in an English-Japanese MT project. The PROLOG version of this parser was inverted, using the method described here, into an efficient PROLOG generator working from regularized parse forms to English sentences. To obtain a PROLOG parser (or any PROLOG program) working in the reverse, requires[1] some manipulation of the clauses, especially the ordering of the literals on their right-hand side, as noted by Dymetman and Isabelle (1988). We do not discuss here certain other transformations used to "normalize" the parser code in order to attain maximum efficiency of the derived generator program (Strzalkowski, 1989).

IN AND OUT ARGUMENTS

Arguments in a PROLOG literal can be marked as either "in" or "out" depending on whether they are bound at the time the literal is submitted for execution or after the computation is completed. For example, in

```
tovo([to,eat,fish],T4,
     [np,[n,john]],P3)
```

the first and the third arguments are "in", while the remaining two are "out". When tovo is used for generation, i.e.,

```
tovo(T1,T4,P1,
     [eat,[np,[n,john]],
          [np,[n,fish]]])
```

then the last argument is "in", while the first and the third are "out"; `T4` is neither "in" nor "out". The information about "in" and "out" status of arguments is important in determining the "direction" in which predicates containing them can be run[2]. As a further example consider the literal

```
subject(A1,A2,WHQ,NUM,P)
```

where `A1` and `A2` are input and output strings of words, `WHQ` indicates whether the subject phrase is a part of a clause within a wh-question, `NUM` is the number of the subject phrase, and `P` is the final translation. During parsing, the "in" arguments are: `A1` and `WHQ`, the "out" arguments are `A2`, `NUM` and `P`; during generation, the "in" arguments are `P` and `WHQ`, the "out" arguments are `A1` and `NUM`. In generating, `A2` is neither "in" nor "out". Thus, upon reversing the direction of computation, an "out" argument does not automatically become an "in" argument, nor does an "in" argument automatically become an "out" argument. Below is a method for computing "in" and "out" status of arguments in any given literal in a PROLOG program, as required by the inversion procedure. This algorithm is already general enough to handle any PROLOG program.

An argument X of literal $pred(\cdots X \cdots)$ on the rhs of a clause is "in" if

(A) it is a constant; or

(B) it is a function and all its arguments are "in"; or

(C) it is "in" or *immediately* "out" in some previous literal $pred_0$ on the rhs of the same clause, i.e., $l(Y) :- pred_0(X,Y), pred(X)$; or

(D) it is "out" in an rhs literal $pred_0$ *delayed until after* some predicate $pred_1$ such that $pred_0$ precedes

[1] Barring the presence of non-reversible operators.

[2] For more discussion on directed predicates in PROLOG see Shoham and McDermott (1984), and Debray (1989).

$pred_1$, and $pred_1$ precedes $pred$ on the rhs;[3] or

(E) it is "in" in the head literal L on lhs of the same clause.

An argument X is "in" in the head literal $L = pred(\cdots X \cdots)$ of a clause if (A), or (B), or

(F) L is the top-level literal and X is "in" in it (known a priori); or

(G) X occurs more than once in L and at least one of these occurrences is "in"; or

(H) for every literal $L_1 = pred(\cdots Y \cdots)$ unifiable with L on the rhs of any clause with the head predicate $pred_1$ different than $pred$, and such that Y unifies with X, Y is "in" in L_1.

We distinguish two categories of "out" arguments in literals appearing on the right-hand side of a clause: *immediate* and *delayed*. An argument X occurring in literal $pred(\cdots X \cdots)$ is *immediately* "out" if it is fully bound[4] immediately after $pred(\cdots X \cdots)$ is executed. An argument X in $pred(\cdots X \cdots)$ is "out" *delayed until after* $pred_0$, if it is fully bound only after $pred_0$, following $pred$ on rhs, is executed. For example, consider the following fragment:

```
vp(SN) :- agree(SN,VN),v(VN).
agree(N,N).
```

If `VN` is immediately "out" in `v`, then `SN` in `agree` is "out" delayed until after `v`. For arguments with their "out" status delayed until after $pred_0$, the "out" status is assigned only after $pred_0$ is executed.

An argument X of literal $pred(\cdots X \cdots)$ on the rhs of a clause is *immediately* "out" if

(A) it is "in" in $pred(\cdots X \cdots)$; or

(B) it is a functional expression and all its arguments are either "in" or immediately "out"; or

(C) for every clause with the head literal $pred(\cdots Y \cdots)$ unifiable with $pred(\cdots X \cdots)$ and such that Y unifies with X, Y is either "in", "out" or "unknwn", and Y is marked "in" or "out" in at least one case.

An argument X of literal $pred(\cdots X \cdots)$ on the rhs of a clause is "out" *delayed until after* $pred_0(\cdots Y \cdots)$ following $pred$ if

(D) Y is immediately "out" in $pred_0$ and $X = f(Y)$; or

(E) X is a functional expression and all of its arguments are either "in" or immediately "out" or "out" delayed until after $pred_0$; or

(F) there is a predicate $pred_1(\cdots X \cdots Z^* \cdots)$ preceding $pred_0$ on the rhs, where Z^* is a subset of arguments of $pred_1$ such that every argument in Z^* is "out" delayed until after $pred_0$ and whenever Z^* is "in" then X is immediately "out" in $pred_1$.

An argument X of literal $pred(\cdots X \cdots)$ on the lhs of a clause is "out" if

(G) it is "in" in $pred(\cdots X \cdots)$; or

(H) it is "out" (immediately or delayed) in literal $pred_1(\cdots X \cdots)$ on the rhs of this clause, providing that $pred_1 \neq pred$ (again, we must take provisions to avoid infinite descend, cf. (H) in "in" algorithm); if $pred_1 = pred$ then X is marked "unknwn".

ESSENTIAL ARGUMENTS

Some arguments of every literal are essential in the sense that the literal cannot be executed successfully unless all of them are bound, at least partially, at the time of execution. A literal may have several alternative, possibly overlapping, sets of essential arguments. If all arguments in any one of such sets of essential arguments are bound, then the literal can be executed. Any set of essential arguments which have the above property is called *essential*. We shall call the set *MSEA* of essential arguments *a minimal set of essential arguments* if it is essential, and no proper subset of *MSEA* is essential. If we alter the ordering of the rhs literals in the definition of a predicate, we may also change its set of *MSEA*'s. We call the set of *MSEA*'s existing for a current definition of a predicate the set of *active MSEA*'s for this predicate. To run a predicate in a certain direction requires that a specific *MSEA* is among the currently active *MSEA*'s for this predicate, and if this is not already the case, then we have to alter the definition of this predicate so as to make this *MSEA* become active. As an example consider the following clause from our PROLOG parser:

```
objectbe(O1,O2,P1,P2,PSA,P) :-
                    venpass(O1,O2,P1,P3),
                    concat([P2,P3],PSA,P).
```

Assuming that {`O1`} and {`P3`} are *MSEA*'s of `venpass` and that `P3` is "out" in `venpass` whenever `O1` is "in", we obtain that {`O1`} is the only candidate for an active *MSEA* in `objectbe`. This is because `P3` is not present on the argument list of `objectbe`, and thus cannot receive a binding before the execution of `venpass` commences. Moving to the `concat` literal, we note that its first argument is partially bound since `P3` is "out" in `venpass`. This is enough for `concat` to execute, and we conclude that `O1` is in fact the only essential argument in `objectbe`. If we reverse the order of `venpass` and `concat`, then {`P`} becomes the new active *MSEA* for `objectbe`, while {`O1`} is no longer active. Given the binding to its third argument, `concat` returns bindings to the

[3] The precedence is with respect to the order of evaluation, which in PROLOG is left-to-right.

[4] An argument is considered *fully bound* if it is a constant or it is bound by a constant; an argument is *partially bound* if it is, or is bound by, a term in which at least one variable is unbound.

first two, and thus it also binds `P3`, which is an essential argument in **venpass**.[5] Below is the general procedure MSEAS for computing the active sets of essential arguments in the head literal of a clause as proposed in (Strzalkowski and Peng, 1990).

Let's consider the following abstract clause defining a predicate R_i:

$$R_i(X_1, \cdots, X_k) :- \quad Q_1(\cdots), \quad Q_2(\cdots), \quad \cdots \quad Q_n(\cdots). \qquad \text{(R)}$$

Suppose that, as defined by (R), R_i has the set $MS_i = \{m_1, \cdots, m_j\}$ of active *MSEA*'s, and let $MR_i \supseteq MS_i$ be the set of all *MSEA* for R_i that can be obtained by permuting the order of literals on the right-hand side of (R). Let us assume further that R_i occurs on rhs of some other clause, as shown below:

$$P(X_1, \cdots, X_n) :- \quad R_1(X_{1,1}, \cdots, X_{1,k_1}), \quad R_2(X_{2,1}, \cdots, X_{2,k_2}), \quad \cdots \quad R_s(X_{s,1}, \cdots, X_{s,k_s}). \qquad \text{(P)}$$

We want to compute *MS*, the set of active *MSEA*'s for P, as defined by (P), where $s \geq 1$, assuming that we know the sets of active *MSEA* for each R_i on the rhs.[6] In the following procedure, the expression $VAR(T)$, where T is a set of terms, denotes the the set of all variables occurring in the terms in T.

MSEAS $(MS, MSEA, VP, i, OUT)$

(1) Start with $VP = VAR(\{X_1, \cdots, X_n\})$, $MSEA = \emptyset$, $i=1$, and $OUT = \emptyset$. When the computation is completed, *MS* is bound to the set of active *MSEA*'s for *P*.

(2) Let MR_1 be the set of active *MSEA*'s of R_1, and let MRU_1 be obtained from MR_1 by replacing all variables in each member of MR_1 by their corresponding actual arguments of R_1 on the rhs of (C1).

(3) If $R_1 = P$ then for every $m_{1,k} \in MRU_1$ if every argument $Y_t \in m_{1,k}$ is *always unifiable*[7] with its corresponding argument X_t in P then remove $m_{1,k}$ from MRU_1. For every set $m_{1,k_j} = m_{1,k} \cup \{X_{1,j}\}$, where $X_{1,j}$ is an argument in R_1 such that it is not already in $m_{1,k}$ and it is not *always unifiable* with its corresponding argument in P, and m_{1,k_j} is not a superset of any other $m_{1,l}$ remaining in MRU_1, add m_{1,k_j} to MRU_1.

(4) For each $m_{1,j} \in MRU_1$ ($j=1 \cdots r_1$) compute $\mu_{1,j} := VAR(m_{1,j}) \cap VP$. Let $MP_1 = \{\mu_{1,j} \mid \phi(\mu_{1,j}), j=1 \cdots r\}$, where $r>0$, and $\phi(\mu_{1,j}) = [\mu_{1,j} \neq \emptyset$ or $(\mu_{1,j} = \emptyset$ and $VAR(m_{1,j}) = \emptyset)]$. If $MP_1 = \emptyset$ then QUIT: (C1) is ill-formed and cannot be executed.

(5) For each $\mu_{1,j} \in MP_1$ we do the following: (a) assume that $\mu_{1,j}$ is "in" in R_1; (b) compute set $OUT_{1,j}$ of "out" arguments for R_1; (c) call $MSEAS(MS_{1,j}, \mu_{1,j}, VP, 2, OUT_{1,j})$; (d) assign MS $:= \bigcup_{j=1..r} MS_{1,j}$.

(6) In some i-th step, where $1<i\leq s$, and $MSEA = \mu_{i-1,k}$, let's suppose that MR_i and MRU_i are the sets of active *MSEA*'s and their instantiations with actual arguments of R_i, for the literal R_i on the rhs of (P).

(7) If $R_i = P$ then for every $m_{i,u} \in MRU_i$ if every argument $Y_t \in m_{i,u}$ is *always unifiable* with its corresponding argument X_t in P then remove $m_{i,u}$ from MRU_i. For every set $m_{i,u_j} = m_{i,u} \cup \{X_{i,j}\}$ where $X_{1,i}$ is an argument in R_i such that it is not already in $m_{i,u}$ and it is not *always unifiable* with its corresponding argument in P and m_{i,u_j} is not a superset of any other $m_{i,l}$ remaining in MRU_i, add m_{i,u_j} to MRU_1.

(8) Again, we compute the set $MP_i = \{\mu_{i,j} \mid j=1 \cdots r_i\}$, where $\mu_{i,j} = (VAR(m_{i,j}) - OUT_{i-1,k})$, where $OUT_{i-1,k}$ is the set of all "out" arguments in literals R_1 to R_{i-1}.

(9) For each $\mu_{i,j}$ remaining in MP_i where $i\leq s$ do the following:
   (a) if $\mu_{i,j} = \emptyset$ then: (i) compute the set OUT_j of "out" arguments of R_i; (ii) compute the union $OUT_{i,j} := OUT_j \cup OUT_{i-1,k}$; (iii) call $MSEAS(MS_{i,j}, \mu_{i-1,k}, VP, i+1, OUT_{i,j})$;
   (b) otherwise, if $\mu_{i,j} \neq \emptyset$ then find all distinct minimal size sets $\nu_t \subseteq VP$ such that whenever the arguments in ν_t are "in", then the arguments in $\mu_{i,j}$ are "out". If such ν_t's exist, then for every ν_t do: (i) assume ν_t is "in" in P; (ii) compute the set OUT_{i,j_t} of "out" arguments in all literals from R_1 to R_i; (iii) call $MSEAS(MS_{i,j_t}, \mu_{i-1,k} \cup \nu_t, VP, i+1, OUT_{i,j_t})$;
   (c) otherwise, if no such ν_t exist, $MS_{i,j} := \emptyset$.

(10) Compute $MS := \bigcup_{j=1..r} MS_{i,j}$;

[5] We note that since `concat` could also be executed with `P2` bound, the set `{O1,P2}` constitutes another active *MSEA* for inverted `objectbe`. However, this *MSEA* is of little use since the binding to `O1` is unlikely to be known in generation.

[6] *MSEA*'s of basic predicates, such as concat, are assumed to be known a priori; *MSEA*'s for recursive predicates are first computed from non-recursive clauses. We assume that symbols X_i in definitions (P) and (R) above represent terms, not just variables. For more details see (Strzalkowski and Peng, 1990). The case of $s=0$ is discussed below.

[7] A term Y is *always unifiable* with a term X if they unify regardless of the possible bindings of any variables occurring in Y (variables standardized apart), while the variables occurring in X are unbound. Any term is always unifiable with a variable, but the inverse is not necessarily true.

(11) For $i=s+1$ set $MS := \{MSEA\}$.

In order to compute the set of all *MSEA*'s for *P*, the procedure presented above need to be modified so that it would consider all feasible orderings of literals on the rhs of (P), using information about all *MSEA*'s for R_i's. This modified procedure would regard the rhs of (P) as an unordered set of literals, and use various heuristics to consider only selected orderings. We outline the modified procedure briefly below.

Let *RR* denote this set, that is, $RR = \{R_i \mid i=1 \cdots s\}$. We add *RR* as an extra argument to MSEAS procedure, so that the call to the modified version becomes *MSEAS*(*MS*,*MSEA*,*VP*,*RR*,*i*,*OUT*). Next we modify step (2) in the procedure as follows:

(2') For every element $R_{t,1} \in RR$, do (2) to (5):

(2) Let $MR_{t,1}$ be the set of all *MSEA*'s of $R_{t,1}$, and let $MRU_{t,1}$ be obtained from $MR_{t,1}$ by replacing all variables in each member of $MR_{t,1}$ by their corresponding actual arguments of $R_{t,1}$.

Further steps are modified accordingly. The reader may note that the modified MSEAS procedure will consider all feasible ways of ordering elements of *RR*. In the steps shown above, we select all literals as potential leading elements on the right hand side, even though most of them will be rejected by steps (3) and (4). For those that survive, we will select elements from the rest of *RR* that can follow them. In step (5) the recursive call to MSEAS will be $MSEAS(MS_{t,1,j},\mu_{t,1,j},VP,RR-\{R_{t,1}\},2,OUT_{t,1,j})$. In step (6), that is, in *i*-th step of the recursion, we consider all elements of $RR-\{R_{t,j} \mid j=1 \cdots i-1\}$, for selection of the *i*-th literal on the right-hand side. By this time we will have already generated a number of possible orderings of $\{R_l \mid l=1 \cdots i-1\}$. We add step (6') which contains the head of an iteration over the remaining elements of *RR*, and covering steps (6) to (11). Again, some of the elements of *RR* will be rejected in steps (7) and (10). We continue until *RR* is completely ordered, possibly in several different ways. For each such possible ordering a set of *MSEA*'s will be computed. Step (12) is an end condition with $RR=\emptyset$. To obtain a meaningful result, *MSEA*'s in $MR_{t,j}$'s must be grouped into sets of these which are active at the same time, that is, they belong to the set of active *MSEA*'s for a specific definition of *P* (i.e., ordering of *RR*). *MSEA*'s belonging to different groups give rise to alternative sets of *MSEA*'s in the final set *MS*. Note that in this modified algorithm, *MS* becomes a set of sets of sets.

An important part in the process of computing essential arguments for literals is the selection of *MSEA*'s for lexicon access and other primitives whose definitions are not subject to change. As an example, consider a fragment of a lexicon:

```
verb([looks|V],V,sg,look).
verb([look|V],V,pl,look).
verb([arrives|V],V,sg,arrive).
verb([arrive|V],V,pl,arrive).
```

The lexicon access primitive `verb(V1,V2,Nm,P)` has two sets of essential arguments: {`V1`} and {`Nm,P`}. This is because {`V1`} can be consistently unified with at most one of {`[looks|V]`}, {`[look|V]`}, {`[arrive|V]`}, etc., at a time. Similarly, {`Nm,P`} can be consistently unified at any one time with at most one of {`sg,look`}, {`pl,look`}, {`sg,arrive`}, etc. Note that neither {`P`} nor {`Nm`} alone are sufficient, since they would unify with corresponding arguments in more than one clause. This indeterminacy, although not necessarily fatal, may lead to severe inefficiency if the generator has to make long backups before a number agreement is established between, say, a verb and its subject. On the other hand, if the representation from which we generate does not include information about the lexical number for constituents, we may have to accept {`P`} as the generation-mode *MSEA* for `verb`, or else we risk that the grammar will not be reversed at all.

REORDERING LITERALS IN CLAUSES

When attempting to expand a literal on the rhs of any clause the following basic rule should be observed: never expand a literal before at least one its active *MSEA*'s is "in", which means that all arguments in at least one *MSEA* are bound. The following algorithm uses this simple principle to reorder rhs of parser clauses for reversed use in generation. This algorithm uses the information about "in" and "out" arguments for literals and sets of *MSEA*'s for predicates. If the "in" *MSEA* of a literal is not active then the rhs's of every definition of this predicate is recursively reordered so that the selected *MSEA* becomes active. We proceed top-down altering definitions of predicates of the literals to make their *MSEA*'s active as necessary, starting with the top level predicate parse(S,P), where P is marked "in" (parse structure) and S is marked "out" (generated sentence). We continue until we reach the level of atomic or non-reversible primitives such as concat, member, or dictionary look-up routines. If this process succeeds at reversing predicate definitions at each level, then the reversed-parser generator is obtained.

INVERSE("head :- old-rhs",ins,outs);
{ins and outs are subsets of VAR(head) which
are "in" and are required to be "out", respectively}
begin
  compute M the set of all *MSEA*'s for head;
  for every *MSEA* m $\in$ M do
  begin
    OUT := $\emptyset$;
    if m is an active *MSEA* such that m$\subseteq$ins then
    begin
      compute "out" arguments in head;
      add them to OUT;

```
    if outs⊆OUT then DONE("head:-old-rhs")
  end
  else if m is a non-active MSEA and m⊆ins then
  begin
    new-rhs := ∅; QUIT := false;
    old-rhs-1 := old-rhs;
    for every literal L do M_L := ∅;
    {done only once during the inversion}
    repeat
      mark "in" old-rhs-1 arguments which are
        either constants, or marked "in" in head,
        or marked "in", or "out" in new-rhs;
      select a literal L in old-rhs-1 which has
        an "in" MSEA m_L and if m_L is not active in L
        then either M_L = ∅ or m_L ∈ M_L;
      set up a backtracking point containing
        all the remaining alternatives
        to select L from old-rhs-1;
      if L exists then
      begin
        if m_L is non-active in L then
        begin
          if M_L = ∅ then M_L := M_L ∪ {m_L};
          for every clause "L1 :- rhs_L1" such that
            L1 has the same predicate as L do
          begin
            INVERSE("L1 :- rhs_L1",M_L,∅);
            if GIVEUP returned then backup, undoing
                all changes, to the latest backtracking
                point and select another alternative
          end
        end;
        compute "in" and "out" arguments in L;
        add "out" arguments to OUT;
        new-rhs := APPEND-AT-THE-END(new-rhs,L);
        old-rhs-1 := REMOVE(old-rhs-1,L)
      end {if}
      else begin
        backup, undoing all changes, to the latest
          backtracking point and select another
          alternative;
        if no such backtracking point exists then
          QUIT := true
      end {else}
    until old-rhs-1 = ∅ or QUIT;
    if outs⊆OUT and not QUIT then
        DONE("head:-new-rhs")
  end {elseif}
 end; {for}
 GIVEUP("grammar can't be inverted as specified")
end;
```

MOVING LITERALS BETWEEN CLAUSES

The inversion algorithm, as realized by the procedure INVERSE, requires that for each clause in the parser code we can find a definite order of literals on its right-hand side that would satisfy the requirements of running this clause in the reverse: appropriate minimal sets of essential arguments (*MSEA*'s) are bound at the right time. However, this requirement is by no means guaranteed and INVERSE may encounter clauses for which no ordering of the literals on the right-hand side would be possible. It may happen, of course, that the clause itself is ill-formed but this is not the only situation. It may be that two or more literals on the right-hand side of a clause cannot be scheduled because each is waiting for the other to deliver the missing bindings to some essential arguments. As an example, consider the grammar fragment below:

```
sent(P)       :- sub(N1,P1),
                 vp(N1,P1,P).
vp(N1,P1,P)   :- v(N2,P2),
                 agree(N1,N2),
                 obj(P1,P2,P).
```

In the generation mode, that is, with the variable `P` instantiated by the parse structure of a sentence, the following active *MSEA*'s and "out" arguments have been computed:

predicate	MSEA	"out"
sent	{P}	
sub	{P1}	N1
vp	{N1,P}	P1
v	{P2}	N2
agree	{N1,N2}	
obj	{P}	P1,P2

In order to use these rules for generation, we would have to change the order of literals on the right-hand side of `sent` clause, so that the `vp` is expanded first. However, doing so would require that the variable `N1` is bound. This we could get by firing `subj` first, but we can't do this either, since we wouldn't know the binding to `P1`. We note, however, that if we consider the two clauses together, then a consistent ordering of literals can be found. To see it, we expand `vp` on the right-hand side of the first clause replacing it with the appropriately unified literals in the right-hand side of the second clause, and obtain a single new clause that can be reordered for generation as follows:

```
sent(P)  :- obj(P1,P2,P),
            v(N2,P2),
            sub(N1,P1),
            agree(N1,N2).
```

Now we can reintroduce the non-terminal `vp`, and break the above rule back into two. Note that as a result `agree` migrated to the first clause, and `N2` replaced `N1` on the argument list of `vp`. Note also that `N2` is not an essential argument in the new `vp`.

```
sent(P)       :- vp(N2,P1,P),
                 sub(N1,P1),
                 agree(N1,N2).
vp(N2,P1,P)   :- obj(P1,P2,P),
                 v(N2,P2).
```

The only thing that remains to be done is to automatically determine the arguments of the new **vp** predicate. Doubtless, it will be a subset of the arguments occurring in the literals that create the right-hand side of the new clause. In the example given this set is {N2,P1,P2,P}. From this set, we remove all those arguments which do not occur in other literals of the original clause, that is, before the break up. The only such argument is P2, and thus the final set of arguments to **vp** becomes {N2,P1,P}, as shown above. The complete algorithm for interclausal reordering of goals can be described by a largely straightforward extension to INVERSE (Strzalkowski, 1989)[8]

CONCLUSIONS

In this paper we presented an algorithm for automatic inversion of a unification parser for natural language into an efficient unification generator. The inverted program of the generator is obtained by an off-line compilation process which directly manipulates the PROLOG code of the parser program. We distinguish two logical stages of this transformation: computing the minimal sets of essential arguments (*MSEA*'s) for predicates, and generating the inverted program code with INVERSE. We have completed a first implementation of the system and used it to derive both a parser and a generator from a single DCG grammar for English (Strzalkowski and Peng, 1990).

This method is contrasted with the approaches that seek to define a generalized but computationally expensive evaluation strategy for running a grammar in either direction without a need to manipulate its rules (Shieber, 1988), (Shieber et al., 1989), and see also (Colmerauer, 1982) and (Naish, 1986) for some relevant techniques, employing the trick known as *goal freezing*. To reduce the cost of the goal freezing, and also to circumvent some of its deficiencies, Shieber et al. (1989) introduce a mixed top-down/bottom-up goal expansion strategy, in which only selected goals are expanded during the top-down phase of the interpreter. This technique, still substantially more expensive than a fixed-order top-down interpreter, does not by itself guarantee that the underlying grammar formalism can be used bidirectionally, and it may need to be augmented by static goal reordering, as described in this paper.

ACKNOWLEDGMENTS

Ralph Grishman, Ping Peng and other members of the Natural Language Discussion Group provided valuable comments to earlier versions of this paper. This paper is based upon work supported by the Defense Advanced Research Project Agency under Contract N00014-85-K-0163 from the Office of Naval Research.

[8] It should be noted that recursive clauses are never used for literal expansion during interclausal ordering, and that literals are not moved to or from recursive clauses, although argument lists of recursive literals may be affected by literals being moved elsewhere.

REFERENCES

Colmerauer, Alain. 1982. *PROLOG II: Manuel de reference et modele theorique.* Groupe d'Intelligence Artificielle, Faculte de Sciences de Luminy, Marseille.

Dymetman, Marc and Isabelle, Pierre. 1988. "Reversible Logic Grammars for Machine Translation." Proc. of the Second Int. Conference on Machine Translation, Pittsburgh, PA.

Debray, Saumya, K. 1989. "Static Inference Modes and Data Dependencies in Logic Programs." *ACM Transactions on Programming Languages and Systems,* 11(3), July 1989, pp. 418-450.

Grishman, Ralph. 1986. *Proteus Parser Reference Manual.* Proteus Project Memorandum #4, Courant Institute of Mathematical Sciences, New York University.

Naish, Lee. 1986. *Negation and Control in PROLOG.* Lecture Notes in Computer Science, 238, Springer.

Shieber, Stuart, M. 1988. "A uniform architecture for parsing and generation." *Proceedings of the 12th COLING,* Budapest, Hungary, pp. 614-619.

Shieber, Stuart M., van Noord, Gertjan, Moore, Robert C. and Pereira, Fernando C.N. 1989. *A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms.* Proceedings of the 27th Meeting of the ACL, Vancouver, B.C., pp. 7-17.

Shoham, Yoav and McDermott, Drew V. 1984. "Directed Relations and Inversion of PROLOG Programs." Proc. of the Int. Conference of Fifth Generation Computer Systems.

Strzalkowski, Tomek. 1989. *Automated Inversion of a Unification Parser into a Unification Generator.* Technical Report 465, Courant Institute of Mathematical Sciences, New York University.

Strzalkowski, Tomek. 1990. "An algorithm for inverting a unification grammar into an efficient unification generator." *Applied Mathematics Letters,* vol. 3, no. 1, pp. 93-96. Pergamon Press.

Strzalkowski, Tomek and Peng, Ping. 1990. "Automated Inversion of Logic Grammars for Generation." *Proceedings of the 28th Annual Meeting of the ACL,* Pittsburgh, PA.