AUTOMATED INVERSION OF LOGIC GRAMMARS FOR GENERATION

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

ABSTRACT

We describe a system of reversible grammar in which, given a logic-grammar specification of a natural language, two efficient PROLOG programs are derived by an off-line compilation process: a parser and a generator for this language. The centerpiece of the system is the inversion algorithm designed to compute the generator code from the parser's PROLOG code, using the collection of minimal sets of essential arguments (*MSEA*) for predicates. The system has been implemented to work with Definite Clause Grammars (DCG) and is a part of an English-Japanese machine translation project currently under development at NYU's Courant Institute.

INTRODUCTION

The results reported in this paper are part of the ongoing research project to explore possibilities of an automated derivation of both an efficient parser and an efficient generator for natural language, such as English or Japanese, from a formal specification for this language. Thus, given a grammar-like description of a language, specifying both its syntax as well as "semantics" (by which we mean a correspondence of well-formed expressions of natural language to expressions of a formal representation language) we want to obtain, by a fully automatic process, two possibly different programs: a parser and a generator. The parser will translate well-formed expression of the source language into expressions of the language of "semantic" representation, such as regularized operator-argument forms, or formulas in logic. The generator, on the other hand, will accept well-formed expressions of the semantic representation language and produce corresponding expressions in the source natural language.

Among the arguments for adopting the bidirectional design in NLP the following are perhaps the most widely shared:

- A bidirectional NLP system, or a system whose inverse can be derived by a fully automated process, greatly reduces effort required for the system development, since we need to write only one program or specification instead of two. The actual amount of savings ultimately depends upon the extend to which the NLP system is made bidirectional, for example, how much of the language analysis process can be inverted for generation. At present we reverse just a little more than a syntactic parser, but the method can be applied to more advanced analyzers as well.
- Using a single specification (a grammar) underlying both the analysis and the synthesis processes leads to more accurate capturing of the language. Although no NLP grammar is ever complete, the grammars used in parsing tend to be "too loose", or unsound, in that they would frequently accept various ill-formed strings as legitimate sentences, while the grammars used for generation are usually made "too tight" as a result of limiting their output to the "best" surface forms. A reversible system for both parsing and generation requires a finely balanced grammar which is sound and as complete as possible.
- A reversible grammar provides, by design, the match between system's analysis and generation capabilities, which is especially important in interactive systems. A discrepancy in this capacity may mislead the user, who tends to assume that what is generated as output is also acceptable as input, and vice-versa.
- Finally, a bidirectional system can be expected to be more robust, easier to maintain and modify, and altogether more perspicuous.

In the work reported here we concentrated on unification-based formalisms, in particular Definite Clause Grammars (Pereira & Warren, 1980), which can be compiled dually into PROLOG parser and generator, where the generator is obtained from the parser's code with the inversion procedure described below. As noted by Dymetman and Isabelle (1988), this transformation must involve rearranging the order of literals on the right-hand side of some clauses. We noted that the design of the string grammar (Sager, 1981) makes it more suitable as a basis of a reversible system than other grammar designs, although other grammars can be "normalized" (Strzalkowski, 1989). We also would like to point out that our main emphasis is on the problem of

reversibility rather than generation, the latter involving many problems that we don't deal with here (see, e.g. Derr & McKeown, 1984; McKeown, 1985).

RELATED WORK

The idea that a generator for a language might be considered as an inverse of the parser for the same language has been around for some time, but it was only recently that more serious attention started to be paid to the problem. We look here only very briefly at some most recent work in unification-based grammars. Dymetman and Isabelle (1988) address the problem of inverting a definite clause parser into a generator in context of a machine translation system and describe a top-down interpreter with dynamic selection of AND goals[1] (and therefore more flexible than, say, left-to-right interpreter) that can execute a given DCG grammar in either direction depending only upon the binding status of arguments in the top-level literal. This approach, although conceptually quite general, proves far too expensive in practice. The main source of overhead comes, it is pointed out, from employing the trick known as *goal freezing* (Colmerauer, 1982; Naish, 1986), that stops expansion of currently active AND goals until certain variables get instantiated. The cost, however, is not the only reason why the goal freezing techniques, and their variations, are not satisfactory. As Shieber et al. (1989) point out, the inherently top-down character of goal freezing interpreters may occasionally cause serious troubles during execution of certain types of recursive goals. They propose to replace the dynamic ordering of AND goals by a mixed top-down/bottom-up interpretation. In this technique, certain goals, namely those whose expansion is defined by the so-called "chain rules"[2], are not expanded during the top-down phase of the interpreter, but instead they are passed over until a nearest non-chain rule is reached. In the bottom-up phase the missing parts of the goal-expansion tree will be filled in by applying the chain rules in a backward manner. This technique, still substantially more expensive than a fixed-order top-down interpreter, does not by itself guarantee that we can use the underlying grammar formalism bidirectionally. The reason is that in order to achieve bidirectionality, we need either to impose a proper static ordering of the "non-chain" AND goals (i.e., those which are not responsible for making a rule a "chain rule"), or resort to dynamic ordering of such goals, putting the goal freezing back into the picture.

In contrast with the above, the parser inversion procedure described in this paper does not require a run-time overhead and can be performed by an off-line compilation process. It may, however, require that the grammar is normalized prior to its inversion. We briefly discuss the grammar normalization problem at the end of this paper.

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[3]. Below we present a simple method for computing "in" and "out" arguments in PROLOG literals.[4]

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 "out" in some previous literal on the rhs of the same clause, i.e., $l(Y) :- r(X,Y), pred(X)$; or (D) 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 (E) L is the top-level literal and X is "in" in it (known a priori); or (F) X occurs more than once in L and at

[1] Literals on the right-hand side of a clause create AND goals; literals with the same predicate names on the left-hand sides of different clauses create OR goals.

[2] A chain rule is one where the main binding-carrying argument is passed unchanged from the left-hand side to the right. For example, `assert(P) --> subj(P1), verb(P2), obj(P1,P2,P)`, is a chain rule with respect to the argument `P`.

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

[4] This simple algorithm is all we need to complete the experiment at hand. A general method for computing "in"/"out" arguments is given in (Strzalkowski, 1989). In this and further algorithms we use abbreviations rhs and lhs to stand for right-hand side and left-hand side (of a clause), respectively.

least one of these occurrences is "in"; or (G) 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.

A similar algorithm can be proposed for computing "out" arguments. We introduce "unknwn" as a third status marker for arguments occurring in certain recursive clauses.

An argument X of literal $pred(\cdots X \cdots)$ on the rhs of a clause is "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 "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 lhs of a clause is "out" if (D) it is "in" in $pred(\cdots X \cdots)$; or (E) it is "out" in literal $pred_1(\cdots X \cdots)$ on the rhs of this clause, providing that $pred_1 \neq pred$;[5] if $pred_1 = pred$ then X is marked "unknwn".

Note that this method predicts the "in" and "out" status of arguments in a literal only if the evaluation of this literal ends successfully. In case it does not (a failure or a loop) the "in"/"out" status of arguments becomes irrelevant.

COMPUTING 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. For example, the predicate `tovo(T1,T4,P1,P3)` that recognizes "to+verb+object" object strings can be executed only if either `T1` or `P3` is bound.[6] [7] If `tovo` is used to parse then `T1` must be bound; if it is used to generate then `P3` must be bound. In general, 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 has the above property is called *essential*. We shall call a set *MSEA* of essential arguments *a minimal set of essential arguments* if it is essential, and no proper subset of *MSEA* is essential.

A collection of minimal sets of essential arguments (*MSEA*'s) of a predicate depends upon the way this predicate is defined. 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. Consider the following abstract clause defining predicate R_i:

$$\begin{aligned} R_i(X_1, \cdots, X_k) :- \quad & \text{(D1)} \\ & Q_1(\cdots), \\ & Q_2(\cdots), \\ & \cdots \\ & Q_n(\cdots). \end{aligned}$$

Suppose that, as defined by (D1), 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 (D1). Let us assume further that R_i occurs on rhs of some other clause, as shown below:

$$\begin{aligned} P(X_1, \cdots, X_n) :- \quad & \text{(C1)} \\ & R_1(X_{1,1}, \cdots, X_{1,k_1}), \\ & R_2(X_{2,1}, \cdots, X_{2,k_2}), \\ & \cdots \\ & R_s(X_{s,1}, \cdots, X_{s,k_s}). \end{aligned}$$

We want to compute *MS*, the set of active *MSEA*'s for P, as defined by (C1), where $s \geq 0$, assuming that we know the sets of active *MSEA* for each R_i on the rhs.[8] If s=0, that is P has no rhs in its definition, then if $P(X_1, \cdots, X_n)$ is a call to P on the rhs of some clause and X^* is a subset of $\{X_1, \cdots, X_n\}$ then X^* is a *MSEA* in P if X^* is the smallest set such that all arguments in X^* consistently unify (at the same time) with the corresponding arguments in at most 1 occurrence of P on the lhs anywhere in the program.[9]

[5] Again, we must take provisions to avoid infinite descend, c.f. (G) in "in" algorithm.

[6] Assuming that `tovo` is defined as follows (simplified): `tovo(T1,T4,P1,P3) :- to(T1,T2), v(T2,T3,P2), object(T3,T4,P1,P2,P3).`

[7] An argument is considered *fully bound* is it is a constant or it is bound by a constant; an argument is *partially bound* if it is, or is bound by, a functional expression (not a variable) in which at least one variable is unbound.

[8] *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.

[9] The *at most 1* requirement is the strictest possible, and it can be relaxed to *at most n* in specific applications. The choice of *n* may depend upon the nature of the input language being processed (it may be n-degree ambiguous), and/or the cost of backing up from unsuccessful calls. For example, consider the words *every* and *all*: both can be translated into a single universal quantifier, but upon generation we face ambiguity. If the representation from

When $s \geq 1$, that is, P has at least one literal on the rhs, we use the recursive procedure MSEAS to compute the set of *MSEA*'s for P, providing that we already know the set of *MSEA*'s for each literal occurring on the rhs. Let T be a set of terms, that is, variables and functional expressions, then $VAR(T)$ is the set of all variables occurring in the terms of T. Thus $VAR(\{f(X),Y,g(c,f(Z),X)\}) = \{X,Y,Z\}$. We assume that symbols X_i in definitions (C1) and (D1) above represent terms, not just variables. The following algorithm is suggested for computing sets of active *MSEA*'s in P where $i \geq 1$.

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* 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.[10]

(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 (C1).

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

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

The procedure presented here can be modified to compute the set of all *MSEA*'s for P by considering all feasible orderings of literals on the rhs of (C1) and using information about all *MSEA*'s for R_i's. This modified procedure would regard the rhs of (C1) as an unordered set of literals, and use various heuristics to consider only selected orderings.

REORDERING LITERALS IN CLAUSES

When attempting to expand a literal on the rhs of any clause the following basic rule should be

which we generate is devoid of any constraints on the lexical number of surface words, we may have to tolerate multiple choices, at some point. Any decision made at this level as to which arguments are to be essential, may affect the reversibility of the grammar.

[10] An argument Y is *always unifiable* with an argument 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. Thus, any term is always unifiable with a variable; however, a variable is not always unifiable with a non-variable. For example, variable X is not always unifiable with $f(Y)$ because if we substitute $g(Z)$ for X then the so obtained terms do not unify. The purpose of including steps (3) and (7) is to eliminate from consideration certain 'obviously' ill-formed recursive clauses. A more elaborate version of this condition is needed to take care of less obvious cases.

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. When reversing a parser, we start with the top level predicate `pars_gen(S,P)` assuming that variable `P` is bound to the regularized parse structure of a sentence. We explicitly identify and mark `P` as "in" and add the requirement that `S` must be marked "out" upon completion of rhs reordering. We proceed to adjust the definition of `pars_gen` to reflect that now {`P`} is an active *MSEA*. We continue until we reach the level of atomic or non-reversible primitives such as concat, member, or dictionary look-up routines. If this top-down process succeeds at reversing predicate definitions at each level down to the primitives, and the primitives need no redefinition, then the process is successful, and the reversed-parser generator is obtained. The algorithm can be extended in many ways, including interclausal reordering of literals, which may be required in some situations (Strzalkowski, 1989).

```
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 ∈ M do
  begin
    OUT := ∅;
    if m is an active MSEA such that m⊆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("can't invert as specified")
end;
```

THE IMPLEMENTATION

We have implemented an interpreter, which translates Definite Clause Grammar dually into a parser and a generator. The interpreter first transforms a DCG grammar into equivalent PROLOG code, which is subsequently inverted into a generator. For each predicate we compute the minimal sets of essential arguments that would need to be active if the program were used in the generation mode. Next, we rearrange the order of the right hand side literals for each clause in such a way that the set of essential arguments in each literal is guaranteed to be bound whenever the literal is chosen for expansion. To implement the algorithm efficiently, we compute the minimal sets of essential arguments and reorder the

literals in the right-hand sides of clauses in one pass through the parser program. As an example, we consider the following rule in our DCG grammar:[11]

```
assertion(S) ->
    sa(S1),
    subject(Sb),
    sa(S2),
    verb(V),
    {Sb:np:number :: V:number},
    sa(S3),
    object(O,V,Vp,Sb,Sp),
    sa(S4),
    {S:verb:head :: Vp:head},
    {S:verb:number :: V:number},
    {S:tense :: [V:tense,O:tense]},
    {S:subject :: Sp},
    {S:object :: O:core},
    {S:sa ::
      [S1:sa,S2:sa,S3:sa,O:sa,S4:sa]}.
```

When translated into PROLOG, it yields the following clause in the parser:

```
assertion(S,L1,L2) :-
    sa(S1,L1,L3),
    subject(Sb,L3,L4),
    sa(S2,L4,L5),
    verb(V,L5,L6),
    Sb:np:number :: V:number,
    sa(S3,L6,L7),
    object(O,V,Vp,Sb,Sp,L7,L8),
    sa(S4,L8,L2),
    S:verb:head :: Vp:head,
    S:verb:number :: V:number,
    S:tense :: [V:tense,O:tense],
    S:subject :: Sp,
    S:object :: O:core,
    S:sa ::
      [S1:sa,S2:sa,S3:sa,O:sa,S4:sa].
```

The parser program is now inverted using the algorithms described in previous sections. As a result, the `assertion` clause above is inverted into a generator clause by rearranging the order of the literals on its right-hand side. The literals are examined from the left to right: if a set of essential arguments is bound, the literal is put into the output queue, otherwise the literal is put into the waiting stack. In the example at hand, the literal `sa(S1,L1,L3)` is examined first. Its *MSEA* is `{S1}`, and since it is not a subset of the set of variables appearing in the head literal, this set cannot receive a binding when the execution of `assertion` starts. It may, however, contain "out" arguments in some other literals on the right-hand side of the clause. We thus remove the first `sa` literal from the clause and place it on hold until its *MSEA* becomes fully instantiated. We proceed to consider the remaining literals in the clause in the same manner, until we reach `S:verb:head :: Vp:head`. One *MSEA* for this literal is `{S}`, which is a subset of the arguments in the head literal. We also determine that `S` is not an "out" argument in any other literal in the clause, and thus it must be bound in `assertion` whenever the clause is to be executed. This means, in turn, that `S` is an essential argument in `assertion`. As we continue this process we find that no further essential arguments are required, that is, `{S}` is a *MSEA* for `assertion`. The literal `S:verb:head :: Vp:head` is output and becomes the top element on the right-hand side of the inverted clause. After all literals in the original clause are processed, we repeat this analysis for all those remaining in the waiting stack until all the literals are output. We add prefix `g_` to each inverted predicate in the generator to distinguish them from their non-inverted versions in the parser. The inverted `assertion` predicate as it appears in the generator is shown below.

```
g_assertion(S,L1,L2) :-
    S:verb:head :: Vp:head,
    S:verb:number :: V:number,
    S:tense :: [V:tense,O:tense],
    S:subject :: Sp,
    S:object :: O:core,
    S:sa ::
      [S1:sa,S2:sa,S3:sa,O:sa,S4:sa],
    g_sa(S4,L3,L2),
    g_object(O,V,Vp,Sb,Sp,L4,L3),
    g_sa(S3,L5,L4),
    Sb:np:number :: V:number,
    g_verb(V,L6,L5),
    g_sa(S2,L7,L6),
    g_subject(Sb,L8,L7),
    g_sa(S1,L1,L8).
```

A single grammar is thus used both for sentence parsing and for generation. The parser or the generator is invoked using the same top-level predicate `pars_gen(S,P)` depending upon the binding status of its arguments: if `S` is bound then the parser is invoked, if `P` is bound the generator is called.

[11] The grammar design is based upon string grammar (Sager, 1981). Nonterminal `sa` stands for a string of sentence adjuncts, such as prepositional or adverbial phrases; `::` is a PROLOG-defined predicate. We show only one rule of the grammar due to the lack of space.

```
| ?- load_gram(grammar).

yes

| ?- pars_gen([jane,takes,a,course],P).

P = [[cat|assertion],
     [tense,present,[]],
     [verb|take],
     [subject,
        [np,[head|jane],
        [number|singular],
        [class|nstudent],
        [tpos],
        [apos],
        [modifier,null]]],
     [object,
        [np,[head|course],
        [number|singular],
        [class|ncourse],
        [tpos|a],
        [apos],
        [modifier,null]]],
     [sa,[],[],[],[],[]]]

yes


| ?- pars_gen(S,
|      [[cat|assertion],
|      [tense,present,[]],
|      [verb|take],
|      [subject,
|         [np,[head|jane],
|         [number|singular],
|         [class|nstudent],
|         [tpos],
|         [apos],
|         [modifier,null]]],
|      [object,
|         [np,[head|course],
|         [number|singular],
|         [class|ncourse],
|         [tpos|a],
|         [apos],
|         [modifier,null]]],
|      [sa,[],[],[],[],[]]]).

S = [jane,takes,a,course]

yes
```

GRAMMAR NORMALIZATION

Thus far we have tacitly assumed that the grammar upon which our parser is based is written in such a way that it can be executed by a top-down interpreter, such as the one used by PROLOG. If this is not the case, that is, if the grammar requires a different kind of interpreter, then the question of invertibility can only be related to this particular type of interpreter. If we want to use the inversion algorithm described here to invert a parser written for an interpreter different than top-down and left-to-right, we need to convert the parser, or the grammar on which it is based, into a version which can be evaluated in a top-down fashion.

One situation where such normalization may be required involves certain types of non-standard recursive goals, as depicted schematically below.

```
vp(A,P) -> vp(f(A,P1),P),compl(P1).
vp(A,P) -> v(A,P).
v(A,P) -> lex.
```

If `vp` is invoked by a top-down, left-to-right interpreter, with the variable `P` instantiated, and if `P1` is the essential argument in `compl`, then there is no way we can successfully execute the first clause, even if we alter the ordering of the literals on its right-hand side, unless, that is, we employ the goal skipping technique discussed by Shieber et al. However, we can easily normalize this code by replacing the first two clauses with functionally equivalent ones that get the recursion firmly under control, and that can be evaluated in a top-down fashion. We assume that `P` is the essential argument in `v(A,P)` and that `A` is "out". The normalized grammar is given below.

```
vp(A,P) -> v(B,P),vp1(B,A).
vp1(f(B,P1),A) -> vp1(B,A),compl(P1).
vp1(A,A).
v(A,P) -> lex.
```

In this new code the recursive second clause will be used so long as its first argument has a form $f(\alpha,\beta)$, where α and β are fully instantiated terms, and it will stop otherwise (either succeed or fail depending upon initial binding to `A`). In general, the fact that a recursive clause is unfit for a top-down execution can be established by computing the collection of minimal sets of essential arguments for its head predicate. If this collection turns out to be empty, the predicate's definition need to be normalized.

Other types of normalization include elimination of some of the chain rules in the grammar, especially if their presence induces undue nondeterminism in the generator. We may also, if necessary, tighten the criteria for selecting the essential arguments, to further enhance the efficiency of the

generator, providing, of course, that this move does not render the grammar non-reversible. For a further discussion of these and related problems the reader is referred to (Strzalkowski, 1989).

CONCLUSIONS

In this paper we presented an algorithm for automated 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. The method described here is contrasted with the approaches that seek to define a generalized but computationally expensive evaluation strategy for running a grammar in either direction without manipulating its rules (Shieber, 1988), (Shieber et al., 1989), (Wedekind, 1989), and see also (Naish, 1986) for some relevant techniques. 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. We note that the present version of INVERSE can operate only upon the declarative specification of a logic grammar and is not prepared to deal with extra-logical control operators such as the cut.

ACKNOWLEDGMENTS

Ralph Grishman and other members of the Natural Language Discussion Group provided valuable comments to earlier versions of this paper. We also thank anonymous reviewers for their suggestions. 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.

REFERENCES

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

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.

Derr, Marcia A. and McKeown, Kathleen R. 1984. "Using Focus to Generate Complex and Simple Sentences." *Proceedings of 10th COLING*, Bonn, Germany, pp. 319-326.

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

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

McKeown, Kathleen R. 1985. *Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text.* Cambridge University Press.

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

Pereira, Fernando C.N. and Warren, David H.D. 1980. "Definite clause grammars for language analysis." *Artificial Intelligence*, 13, pp. 231-278.

Sager, Naomi. 1981. *Natural Language Information Processing.* Addison-Wesley.

Shieber, Stuart M. 1988. "A uniform architecture for parsing and generation." *Proceedings of the 12th COLING*, Budapest, Hungary (1988), 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, Department of Computer Science, 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.

Wedekind, Jurgen. 1988. "Generation as structure driven derivation." *Proceedings of the 12th COLING*, Budapest, Hungary, pp. 732-737.