A GENERAL COMPUTATIONAL METHOD FOR GRAMMAR INVERSION

Tomek Strzalkowski
Courant Institute of Mathematical Sciences
New York University
715 Broadway, rm. 704
New York, NY 10003
tomek@cs.nyu.edu

ABSTRACT

A reversible grammar is usually understood as a computational or linguistic system that can be used both for analysis and generation of the language it defines. For example, a directive *pars_gen (Sent,Form)* would assign, depending upon the binding status of its arguments, the representation *in(Toronto,chased(Fido,John))* to the sentence *Fido chased John in Toronto*, or it would produce one of the several possible paraphrases of this sentence given its representation. Building such bi-directional systems has long been considered critical for various natural language processing tasks, especially in machine translation. This paper presents a general computational method for automated inversion of a unification-based parser for natural language into an efficient generator. It clarifies and expands the results of earlier work on reversible grammars by this author and the others. A more powerful version of the grammar inversion algorithm is developed with a special emphasis being placed on the proper treatment of recursive rules. The grammar inversion algorithm described here is at the core of the Japanese-English machine translation project currently under development at NYU.

REVERSIBLE GRAMMARS

A reversible grammar is usually understood as a computational or linguistic system that can be used both for analysis and generation of the language it defines. For example, a directive *pars_gen (Sent,Form)* would assign, depending upon the binding status of its arguments, the representation *in(Toronto,chased(Fido,John))* to the sentence *Fido chased John in Toronto*, or it would produce one of the several possible paraphrases of this sentence given its representation. In the last several years, there have been a growing amount of research activity in reversible grammars for natural language, particularly in connection with machine translation work, and in natural language generation. Development of reversible grammar systems is considered desirable for variety of reasons that include their immediate use in both parsing and generation, a reduction in the development and maintenance effort, soundness and completeness of linguistic coverage, as well as the match between their analysis and synthesis capabilities. These properties are important in any linguistic system, especially in machine translation, and in various interactive natural language systems where the direction of communication frequently changes. In this paper we are primarily interested in the computational aspects[1] of reversibility that include bi-directional evaluation and dual compilation of computer grammars, inversion of parsers into efficient generators, and derivation of "generating-versions" of existing parsing algorithms. Some of the recent research in this area is reported in (Calder et al., 1989; Dymetman and Isabelle, 1988; Dymetman et al., 1990; Estival, 1990; Hasida and Isizaki, 1987; Ishizaki, 1990; Shieber, 1988; Shieber et al., 1990; Strzalkowski, 1990a-c; Strzalkowski and Peng, 1990; van Noord, 1990; and Wedekind, 1988). Dymetman and Isabelle (1988) describe a top-down interpreter for definite clause grammars that statically reorders clause literals according to a hand-coded specification, and further allows for dynamic selection of AND goals[2] during execution, using the technique known as the *goal freezing* (Colmerauer, 1982; Naish, 1986). Shieber et al. (1990) propose a mixed top-down/bottom-up interpretation, in which certain goals, namely those whose expansion is defined by the so-called "chain rules",[3] 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

[1] For linguistic aspects of reversible grammars, see (Kay, 1984; Landsbergen, 1987; Neuman, 1990; Steedman, 1987).

[2] 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.

[3] A chain rule is one where the main binding-carrying argument (the "head") 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*, assuming that *P* is the 'head' argument.

the chain rules in a backward manner. This technique, known as 'head-driven' evaluation, can be applied quite profitably to various grammar compilation tasks, including the inverse computation, but it requires that the underlying grammar is given in a form where the information about the semantic heads in nonterminals is made explicit. In addition, the procedure, as described in (Shieber et al, 1990), makes no attempt to impose a proper ordering of the "non-chain" goals, which may have an adverse effect on the generator efficiency.[4]

The grammar inversion method described in this paper transforms one set of PROLOG clauses (representing a parser, eg.) into another set of clauses (representing a generator) using an off-line compilation process. The generator is thus just another PROLOG program that has the property of being an inverse of the parser program, that is, it performs inverse computation.[5] A unification grammar is normally compiled into PROLOG to obtain an executable program (usually a parser). Subsequently, the inversion process takes place at the PROLOG code level, and is therefore independent of any specific grammar formalism used. The obtained inverted program has been demonstrated to be quite efficient, and we noted that the same technique can be applied to parser/generator optimization. Our method is also shown to deal adequately with recursive clauses that created problems in purely top-down compilation.[6] The inter-clausal inversion procedure discussed here effects global changes in goal ordering by moving selected goals between clauses and even creating new clauses. The net effect is similar to that achieved in the head-driven evaluation, except that no explicit concept of 'head' or 'chain-rule' is used. The algorithm has been tested on a substantial coverage PROLOG grammar for English derived form the PROTEUS Parser Grammar (Grishman, 1986), and the Linguistic String Grammar for English (Sager, 1981).[7]

[4] Some concern has also been voiced (Gardent and Plainfosse, 1990) about the termination conditions of this algorithm.

[5] Some programs may in fact be multi-directional, and therefore may have several 'inverses' or 'modes'.

[6] Shieber et al. (1990) have shown that some recursive clauses cannot be executed using top-down evaluation thus motivating the use of a mixed top-down/bottom-up evaluation of their 'head-driven' compilation.

[7] At present the grammar consists of 400+ productions.

IN AND OUT ARGUMENTS IN LITERALS

Literals in the grammar clauses can be marked for the "modes" in which they are used. When a literal is submitted to execution then those of its arguments which are bound at that time are called the "in" arguments. After the computation is complete, some of the previously unbound arguments may become bound; these are called the "out" arguments. For example, in *concat*([a,b],[c,d],Z), which is used for list concatenation, the first two arguments are "in", while the third is "out". The roles are reversed when *concat* is used for decomposition, as in *concat*(X,Y,[a,b,c,d]). In the literal *subject(A1,A2,NUM,P)*, taken from an English grammar, *A1* and *A2* are input and output strings of words, *NUM* is the number of the subject phrase, and *P* is the final translation. When the grammar is used for parsing, the "in" argument is *A1*; the "out" arguments are *A2*, *NUM* and *P*; when it is used for generation, the "in" argument is *P*; the "out" arguments are *A1* and *NUM*. In generation, *A2* is neither "in" nor "out".

"In" and "out" status of arguments in a PROLOG program can be computed statically at compile time. The general algorithm has been described in (Strzalkowski, 1990c; Strzalkowski and Peng, 1990).

ESSENTIAL ARGUMENTS: AN EXTENSION

The notion of an essential argument in a PROLOG literal has been first introduced in (Strzalkowski, 1989), and subsequently extended in (Strzalkowski, 1990bc; Strzalkowski and Peng, 1990). In short, X is an essential argument in a literal $p(\cdots X \cdots)$ if X is required to be "in" for a successful evaluation of this literal. By a successful evaluation of a literal we mean here the execution that is guaranteed to stop, and moreover, that will proceed along an optimal path. For instance, an evaluation of the goal *mem*(*a*,*L*), with an intention to find a list *L* of which *a* is a member, leads to a non-terminating execution unless *L*'s value is known. Likewise, a request to generate a main verb in a sentence when the only information we have is its root form (or "logical form") may lead to repeated access to the lexicon until the "correct" surface form is chosen. Therefore, for a lexicon access goal, say *acclex*(*Word*,*Feats*,*Root*), it is reasonable to require that both *Feats* and *Root* are the essential arguments, in other words, that the set {*Feat*,*Root*} is a *minimal set of essential arguments*, or a *MSEA*, for *acclex*. The following procedure computes the set of active

MSEA's in a clause head literal.[8]

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

[computing active MSEAs]

Given a clause $p(X_1, \cdots, X_n) :- r_1(X_{1,1} \cdots X_{1,k_1}), \cdots, r_s(X_{s,1} \cdots X_{s,k_s})$, where $i \geq 1$, we compute the set of active MSEAs in the head predicate p as follows:[9]

(1) Start with $MSEA = \emptyset$, $VP = VAR(\{X_1, \cdots, X_n\})$, $i=1$, and $OUT = OUT_0 = \emptyset$. The set of active *MSEA*'s for p is returned in *MS*.

(2) For $i=1, \cdots, s$, let MR_i be the set of active *MSEA*'s of r_i, and let $MRU_i = \{m_{i,j} \mid j=1 \cdots r_i\}$ be obtained from MR_i by replacing all variables by their corresponding actual arguments of r_i.

(3) 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}.

(4) For each $\mu_{i,j}$ in MP_i where $1 \leq i \leq s$ do the following:

  (a) if $\mu_{i,j} = \emptyset$ then:

    (i) compute set OUT_j of "out" arguments of r_i;

    (ii) compute $OUT_{i,j} := OUT_j \cup OUT_{i-1,k}$;

    (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, $MS_{i,j} := \emptyset$.

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

(6) For $MSEAS(MS,MSEA,VP,s+1,OUT)$, i.e., for $i=s+1$, do $MS := \{MSEA\}$.

As a simple example consider the following clause:

sent(P) :- vp(N,P),np(N).

Assuming that MSEA's for *vp* and *np* are $\{P\}$ and $\{N\}$, respectively, and that N is "out" in *vp*, we can easily compute that $\{P\}$ is the MSEA in *sent*. To see it, we note that MRU_1 for *vp* is $\{\{P\}\}$ and, therefore, that $\mu_{1,1} = \{P\}$. Next, we note that MRU_2 for *np* is $\{\{N\}\}$, and since $OUT_{1,1}$ from *vp* is $\{N\}$, we obtain that $\mu_{2,1} = \emptyset$, and subsequently that $\{P\}$ is the only MSEA in *sent*.

The procedure presented above is sufficient in many cases, but it cannot properly handle certain types of recursive definitions. Consider, for example, the problem of assigning the set of *MSEA*'s to *mem(Elem,List)*, where *mem* (list membership) is defined as follows:

mem (Elem, [First | List]) :-
mem (Elem,List).
mem (Elem, [Elem | List]).

The MSEAS procedure assigns $MS=\{\{Elem\},\{List\}\}$, we note however, that the first argument of *mem* cannot alone control the recursion in the first clause since the right-hand side (rhs) literal would repeatedly unify with the clause head, thus causing infinite recursion. This consideration excludes $\{Elem\}$ from the list of possible *MSEA*s for *mem*. In (Strzalkowski, 1989) we introduced the directed relation *always unifiable* among terms, which was informally characterized as follows. A term X is *always unifiable* with term Y if they unify regardless of any bindings that may occur in X, providing that variables in X and Y are standardized apart, and that Y remains unchanged. According to this definition any term is always unifiable with a variable, while the opposite is not necessarily true. For example, the variable X is not always unifiable with the functional term f(Y) because binding X with g(Z) will make these two terms non-unifiable. This relation can be formally characterized as follows: given two terms X and Y we say that Y is *always unifiable* with X (and write $X \leq Y$) iff the unification of X and Y yields Y, where the variables occurring in X and Y have been standardized apart.[10] Since $\leq$ describes a partial order among terms, we can talk of its transitive closure $\leq^*$. Now we can augment the MSEAS procedure with the following two steps (to be placed between steps (2) and

[8] Active *MSEA*'s are those existing with a given definition of a predicate. Other, non-active *MSEA*'s can be activated when the clauses making up this definition are altered in some way. The procedure can be straightforwardly augmented to compute all *MSEA*s (Strzalkowski, 1990c).

[9] For $i=1$ the sets of essential arguments are selected so as to minimize the number of possible solutions to 1.

[10] So defined, the relation *always unifiable* becomes an inverse of another relation: *less instantiated*, hence the particular direction of $\leq$ sign.

(3)) that would exclude certain *MSEAs* from recursive clauses.

(2A)

If $r_i = p$ then for every $m_{i,u} \in MRU_i$ if for every argument $Y_l \in m_{i,u}$, where Y_l is the l-th argument in r_i, and X_l is the l-th argument in p, we have that $X_l \leq^* Y_l$ then remove $m_{i,u}$ from MRU_i.

(2B)

For every set $m_{i,u_j} = m_{i,u} \cup \{Z_{i,j}\}$, where $Z_{i,j}$ is the j-th argument in r_i such that it is not already in $m_{i,u}$ and it is not the case that $Y_j \leq^* Z_{i,j}$, where Y_j is a j-th argument in p, if m_{i,u_j} is not a superset of any other $m_{i,l}$ remaining in MRU_i, then add m_{i,u_j} to MRU_1.

In order for the MSEAS procedure to retain its practical significance we need to restrict the closure of $\leq$ to be defined only on certain special sets of terms that we call *ordered series*.[11] It turns out that this restricted relation is entirely sufficient in the task of grammar inversion, if we assume that the original grammar is itself well-defined.

DEFINITION 1 (argument series)
Let $p(\cdots Y_0 \cdots) :- r_1, \cdots, r_n$ be a clause, and $r_{i_1}, \cdots, r_{i_k}$ be an ordered subset of the literals on the right-hand side of this clause. Let $r_{i_{k+1}}$ be either a literal to the right of r_{i_k} or the head literal p. The ordered set of terms $<Y_0, X_1, Y_1, \cdots, X_k, Y_k, X_{k+1}>$ is an *argument series* iff the following conditions are met:

(1) X_{k+1} is an argument in $r_{i_{k+1}}$;

(2) for every $i=1 \cdots k$, X_i is different from any X_j for $j<i$;

(3) for every $j=1 \cdots k$, X_j and Y_j are arguments to r_{i_j}, that is, $r_{i_j}(\cdots X_j, Y_j \cdots)$, such that if X_j is "in" then Y_j is "out" [12]; and

(4) for every $j=0 \cdots k$, either $X_{j+1}=Y_j$ or $X_{j+1}=f(Y_j)$ or $Y_j=f(X_{j+1})$, where $f(X)$ denotes a term containing a subterm X.

Note that this definition already ensures that the argument series obtained between X_0 and X_{k+1} is the shortest one. As an example, consider the following clauses:

$$vp(X) :- np(X,Y), vp(Y).$$
$$np(f(X),X).$$

Assuming that the argument X in the literal $vp(X)$ on the left-hand side (lhs) of the first clause is "in", we can easily check that $<X,X,Y,Y>$ constitutes an argument series between arguments of vp in the first clause.

DEFINITION 2 (weakly ordered series)[13]
An argument series $<Y_0, X_1, Y_1, \cdots, X_k, Y_k X_{k+1}>$ in the clause $p :- r_1 \cdots r_n$ is weakly ordered iff $Y_0 \leq^* X_{k+1}$ [or $X_{k+1} \leq^* Y_0$], where $\leq^*$ is a closure of $\leq$ defined as follows:

(1) for every $i=1 \cdots k$, such that $r_{i_j}(\cdots X_j, Y_j \cdots)$ there exists a clause $r_{i_j}(\cdots, X, Y, \cdots) :- s_1, \cdots, s_m$, where X and Y unify with X_j and Y_j, respectively, such that $X \leq^* Y$ [or $Y \leq^* X$];

(2) for every $i=0 \cdots k$, $X_{i+1}=Y_i$ or $X_{i+1}=f(Y_i)$ [or $Y_i=f(X_{i+1})$].

Looking back at the definition of *mem* (*Elem*,*List*) we note that the first (recursive) clause contains two ordered series. The first series, $<Elem,Elem>$, is not ordered (or we may say it is ordered weakly in both directions), and therefore *Elem* on the left-hand side of the clause will always unify with *Elem* on the right, thus causing non-terminating recursion. The other series, $<[First \mid List], List>$, is ordered in such a way that $[First \mid List]$ will not be always unifiable with *List*, and thus the recursion is guaranteed to terminate. This leaves {*List*} as the only acceptable MSEA for *mem*.

Consider now the following new example:

$$vp(X) :- np(X,Y), vp(Y).$$
$$vp(X) :- v(X).$$
$$np(X, f(X)).$$

Note that the series $<X,X,Y,Y>$ in the first clause is ordered so that $X \leq^* Y$. In other words, Y in vp on the rhs is *always unifiable* with X on the lhs. This means that a non-terminating recursion will result if we attempt to execute the first clause top-down. On the other hand, it may be noted that since the series is ordered in one direction only, that is, we don't have $Y \leq^* X$, we could invert it so as to obtain $Y \leq^* X$, but not $X \leq^* Y$. To accomplish this, it is enough to swap the arguments in the clause defining *np*, thus redirecting the recursion. The revised program is guaranteed to

[11] A similar concept of *guide-structure* is introduced in (Dymetman et al., 1990), however the ordered series is less restrictive and covers a larger class of recursive programs.

[12] Y_j may be partially "out"; see (Strzalkowski, 1990c) for the definition of delayed "out" status.

[13] A series can also be strongly ordered in a given direction, if it is weakly ordered in that direction and it is not weakly ordered in the opposite direction.

terminate, providing that *vp*'s argument is bound, which may be achieved by further reordering of goals.[14]

The ordered series relation is crucial in detecting and removing of non-terminating left-recursive rules of the grammar. The first of the following two algorithms finds if an argument series is ordered in a specified direction, without performing a partial evaluation of goals. The second algorithm shows how a directed series can be inverted.

ALGORITHM 1 (finding if $Y_0 \leq^* X_{k+1}$ (weakly))
Given an argument series $<Y_0,X_1,Y_1,\cdots,X_k,Y_kX_{k+1}>$ do the following:

(1) Find if for every $i=0\cdots k$, either $X_{i+1}=Y_i$ or $X_{i+1}=f(Y_i)$; if the answer is negative, return NO and quit.

(2) For every $i=1\cdots k$, find a clause $r_{i_j}(\cdots X,Y\cdots) :- s_1,\cdots,s_m$ such that X_j and Y_j unify with X and Y, respectively, and there is a leading series $<X\cdots Y>$ such that $X \leq^* Y$. Return NO if no such clause is found, and quit.

(3) In the special case when $k=0$, i.e., p has no right-hand side, $Y_0 \leq^* X_1$ if either $Y_0=X_1$ or $X_1=f(Y_0)$. If this is not the case return NO, and quit.

(4) Otherwise, return YES.

When ALGORITHM 1 returns a YES, it has generated an ordered path (i.e., the series with all the necessary subseries) between X_0 and X_{k+1} to prove it. If this path is ordered in one direction only, that is, there exists at least one pair of adjacent elements X_i and Y_j within this path such that either $X_i=f(Y_j)$ or $Y_j=f(X_i)$, but not $X_i=Y_j$, then we say that the path is properly ordered. In addition, if we force ALGORITHM 1 to generate all the paths for a given series, and they all turn out to be properly ordered, then we will say that the series itself is properly ordered. We can attempt to invert a properly ordered path, but not the one which is only improperly ordered, i.e., in both directions. Therefore, for a series to be invertible all its paths must be properly ordered, though not necessarily in the same direction.[15]

ALGORITHM 2 (inverting properly ordered series)
Given a clause $p :- r_1,\cdots,r_n$, and an argument series $<Y_0,X_1,Y_1,\cdots,X_k,Y_kX_{k+1}>$ such that it is properly (weakly) ordered as $X_0 \leq^* X_{k+1}$ [or $X_{k+1} \leq^* X_0$], invert it as follows:

(1) For each $r_{i_j}(\cdots,X_j,Y_j,\cdots)$ appearing on the rhs of the clause, find all clauses $r_{i_j}(\cdots,X,Y,\cdots) :- s_1,\cdots,s_m$ such that X and Y unify with X_j and Y_j, respectively, and there is a proper ordering $X \leq^* Y$ [or $Y \leq^* X$].

(2) Recursively invert the series $<X\cdots Y>$; for the special case where $m=0$, that is, r_{i_j} clause has no rhs, exchange places of X and Y.

(3) For every pair of Y_i and X_{i+1} ($i=0\cdots k$), if either $Y_i=f(X_{i+1})$ or $X_{i+1}=f(Y_i)$, where f is fully instantiated, exchange Y_i with X_{i+1}, and do nothing otherwise.

We now return to the MSEAS procedure and add a new step (2C), that will follow the two steps (2A) and (2B) discussed earlier. The option in (2C) is used when the expansion of a *MSEA* rejected in step (2A) has failed in (2B). In an earlier formulation of this procedure an empty *MSEA* was returned, indicating an non-executable clause. In step (2C) we attempt to rescue those clauses in which the recursion is based on invertible weakly ordered series.

(2C)
Find an argument $Y_t \in m_{i,u}$, a t-th argument of r_i, such that $X_t \leq^* Y_t$, where X_t is the t-th argument in the head literal p and the series $<X_t\cdots Y_t>$ is properly ordered. If no such Y_t is found, augment $m_{i,u}$ with additional arguments; quit if no further progress is possible.[16] Invert the series with ALGORITHM 2, obtaining a strongly ordered series $<X'_t\cdots Y'_t>$ such that $Y'_t \leq^* X'_t$. Replace Y_t with Y'_t in $m_{i,u}$ and add the resulting set to MRU_i.

At this point we may consider a specific linguistic example involving a generalized left-recursive production based on a properly ordered series.[17]

```
[1] sent(V1,V3,Sem) :-
        np(V1,V2,Ssem),
        vp(V2,V3,[Ssem],Sem).
[2] vp(V1,V3,Args,Vsem) :-
        vp(V1,V2,[Csem|Args],Vsem),
        np(V2,V3,Csem).
```

[14] Reordering of goals may be required to make sure that appropriate essential arguments are bound.

[15] Recursion defined with respect to improperly ordered series is considered ill-formed.

[16] As in step (2B) we have to maintain the minimality of $m_{i,u}$.

[17] This example is loosely based on the grammar described in (Shieber et al., 1990).

[3] *vp(V1,V2,Args,Vsem)* :-
v(V1,V2,Args,Vsem).
[4] *v(V1,V2,[Obj,Subj],chased(Subj,Obj))* :-
chased(V1,V2).
[5] *chased([chased |X],X)*.
[6] *np([john |X],X,john)*.
[7] *np([fido |X],X,fido)*.

We concentrate here on the clause [2], and note that there are three argument series between the *vp* literals: <*V1,V1*>, <*Args,[Csem |Args]*>, and <*Vsem,Vsem*>, of which only the second one is invertible. We also note that in clause [3], the collection of *MSEA*s for *vp* include {*V1*} and {*Vsem*}, where *V1* represents the surface string, and *Vsem* its "semantics". When we use this grammar for generation, {*V1*} is eliminated in step (2A) of the MSEAS procedure, while {*Vsem*}, is rescued in step (2C), where it is augmented with *Args* which belongs to the invertible series. We obtain a new set {*Args'*,*Vsem*}, which, if we decide to use it, will also alter the clause [2] as shown below.[18]

[2a] *vp(V1,V3,[Csem |Args],Vsem)* :-
vp(V1,V2,Args,Vsem),np(V2,V3,Csem).

This altered clause can be used in the generator code, but we still have to solve the problem of having the *[Csem |Args]* bound, in addition to *Vsem*.[19] It must be noted that we can no longer meaningfully use the former "in" status (if there was one) of this argument position, once the series it heads has been inverted. We shall return to this problem shortly.

INTRA-CLAUSAL INVERSION

The following general rule is adopted for an effective execution of logic programs: never expand a goal before at least one of its active *MSEA*s is "in". This simple principle can be easily violated when a program written to perform in a given direction is used to run "backwards", or for that matter, in any other direction. In particular, a parser frequently cannot be used as a generator without violating the *MSEA*-binding rule. This problem is particularly acute within a fixed-order evaluation strategy, such as that of PROLOG. The most unpleasant consequence of disregarding the above rule is that the program may go into an infinite loop and have to be aborted, which happens surprisingly often for non-trivial size programs. Even if this does not happen, the program performance can be seriously hampered by excessive guessing and backtracking. Therefore, in order to run a parser in the reverse, we must rearrange the order in which its goals are expanded. This can be achieved in the following three steps:

PROCEDURE INVERSE

(1) Compute "in" and "out" status of arguments for the reversed computation. If the top-level goal *parse(String,Sem)* is used to invoke a generator, then *Sem* is initially "in", while *String* is expected to have "out" status.

(2) Compute sets of all (active and non-active) *MSEA*s for predicates used in the program.

(3) For each goal, if none of its *MSEA*s is "in" then move this goal to a new position with respect to other goals in such a way that at least one of its *MSEA*s is "in". If this "in" *MSEA* is not an active one, recursively invert clauses defining the goal's predicate so as to make the *MSEA* become active.

In a basic formulation of the inversion algorithm the movement of goals in step (3) is confined to be within the right-hand sides of program clauses, that is, goals cannot be moved between clauses. The inversion process proceeds top-down, starting with the top-level clause, for example *parse(String,Sem)* :- *sent(String,[],Sem)*. The restricted movement inversion algorithm INVERSE has been documented in detail in (Strzalkowski, 1990ac). It is demonstrated here on the following clause taken from a parser program, and which recognizes yes-no questions:

yesnoq(A1,A4,P) :-
verb(A1,A2,Num,P2),
subject(A2,A3,Num,P1),
object(A3,A4,P1,P2,P).

When rewriting this clause for generation, we would place *object* first (it has P "in", and A3, P1, P2 "out"), then *subject* (it has the essential P1 "in", and A2 and Num "out"), and finally *verb* (its *MSEA* is either {A1} or {Num,P2}, the latter being completely "in" now). The net effect is the following generator clause:[20]

yesnoq(A1,A4,P) :-
object(A3,A4,P1,P2,P),
subject(A2,A3,Num,P1),
verb(A1,A2,Num,P2).

INVERSE works satisfactorily for most grammars, but it cannot properly handle certain types of clauses

[18] In our inversion algorithm we would not alter the clause until we find that the MSEA needs to be used.

[19] *Vsem* is expected to be "in" during generation, since it carries the "semantics" of *vp*, that is, provides the input to the generator.

[20] Note that the surface linguistic string is not generated from the left to the right.

where no definite ordering of goals can be achieved even after redefinition of goal predicates. This can happen when two or more literals wait for one another to have bindings delivered to some of their essential arguments. The extended MSEAS procedure is used to define a general inversion procedure INTER-CLAUSAL to be discussed next.

INTER-CLAUSAL INVERSION

Consider again the example given at the end of the section on essential arguments. After applying MSEAS procedure we find that the only way to save *MSEA* {*Args*,*Vsem*} is to invert the series <*Args*, [*Csem* | *Args*]> between *vp* literals. This alters the affected clause [2] as shown below (we show also other clauses that will be affected at a later stage):[21]

[1] *sent*(*Sem*) :-
    np(*Ssem*),*vp*([*Ssem*],*Sem*).
[2] *vp*([*Csem* | *Args*],*Vsem*) :-
    vp(*Args*,*Vsem*),*np*(*Csem*).
[3] *vp*(*Args*,*Vsem*) :-
    v(*Args*,*Vsem*).

In order to use the second clause for generation, we now require [*Csem*|*Args*] to be "in" at the head literal *vp*. This, however, is not the case since the only input we receive for generation is the binding to *Sem* in clause [1], and subsequently, *Vsem* in [2], for example, ?–*sent*(*chased*(*Fido*,*John*)). Therefore the code still cannot be executed. Moreover, we note that clause [1] is now deadlocked, since neither *vp* nor *np* can be executed first.[22] At this point the only remaining option is to use interclausal ordering in an effort to inverse [1]. We move *v* from the rhs of [3] to [1], while *np* travels from [1] to [3]. The following new code is obtained (the second argument in the new *vp'* can be dropped, and the new *MSEA* for *vp'* is {*Args*}):[23]

[1'] *sent*(*Sem*) :-
    v(*Args*,*Sem*),*vp'*(*Args*).
[2'] *vp'*([*Csem* | *Args*]) :-
    vp'(*Args*),*np*(*Csem*).
[3'] *vp'*([*Ssem*]) :-
    np(*Ssem*).

This code is executable provided that *Sem* is bound in *sent*. Since *Args* is "out" in *v*, the recursion in [2'] is well defined at last. The effect of the interclausal ordering is achieved by adopting the INTERCLAUSAL procedure described below. The procedure is invoked when a deadlocked clause has been identified by INVERSE, that is, a clause in which the right-hand side literals cannot be completely ordered.

PROCEDURE INTERCLAUSAL(DLC)

[Inter-clausal inversion]

(1) Convert the deadlocked clause into a special canonical form in which the clause consists exclusively of two types of literals: the unification goals in the form $X=Y$ where X is a variable and Y is a term, and the remaining literals whose arguments are only variables (i.e., no constants or functional terms are allowed). Any unification goals derived from the head literal are placed at the front of the rhs. In addition, if $p(\cdots X \cdots)$ is a recursive goal on the rhs of the clause, such that X is an "in" variable unifiable with the head of an inverted series in the definition of p, then replace X by a new variable $X1$ and insert a unification goal $X1=X$. The clause in [1] above is transformed into the following form:

    [1] *sent*(*Sem*) :-
        np(*Ssem*),
        Args=[*Ssem*],
        vp(*Args*,*Sem*).

(2) Select one or more non-unification goals, starting with the "semantic-head" goal (if any), for static expansion. The "semantic-head" goal is the one that shares an essential argument with the literal at the head of the clause. Recursive clauses in the definitions of goal predicates should never be used for expansion. In the example at hand, *vp* can be expanded with [3].

(3) Convert the clauses to be used for goal expansion into the canonical form. In our example [3] needs no conversion.

(4) Expand deadlocked goals by replacing them with appropriately aliased right-hand sides of the clauses selected for expansion. In effect we perform a partial evaluation of these goals. Expanding *vp* in [1] with [3] yields the following new

---

[21] The string variables V1, V2, etc. are dropped for clarity.

[22] There are situations when a clause would not appear deadlocked but still require expansion, for example if we replace [1] by *sent(Sem,Ssem)* :- *vp(Ssem,Sem)*, with *Ssem* bound in *sent*. This clause is equivalent to *sent(Sem,Ssem)* :- *Vsem=Ssem,vp(Vsem,Sem)*, but since the series in [2] has been inverted we can no longer meaningfully evaluate the rhs literals in the given order. In fact we need to evaluate *vp* first which cannot be done until *Vsem* is bound.

[23] An alternative is to leave [1] intact (except for goal ordering) and add an "interface" clause that would relate the old *vp* to the new *vp'*. In such case the procedure would generate an additional argument for *vp'* in order to return the final value of *Args* which needs to be passed to *np*.

clause:

[1a] *sent*(*Sem*) :-
np(*Ssem*),
Args=[*Ssem*],
v(*Args*,*Sem*).

(5) Find an executable order of the goals in the expanded clause. If not possible, expand more goals by recursively invoking INTERCLAUSAL, until the clause can be ordered or no further expansion is possible. In our example [1a] can be ordered as follows:

[1b] *sent*(*Sem*) :-
v(*Args*,*Sem*),
Args=[*Ssem*],
np(*Ssem*).

(6) Break the expanded clause back into two (or more) "original" clauses in such a way that: (a) the resulting clauses are executable, and (b) the clause which has been expanded is made as general as possible by moving as many unification goals as possible out to the clause(s) used in expansion. In our example *v*(*Args*,*Sem*) has to remain in [1b], but the remainer of the rhs can be moved to the new *vp'* clause. We obtain the following clauses (note that clause [2] has thus far remained unchanged throughout this process):

[1b] *sent*(*Sem*) :-
v(*Args*,*Sem*),
vp'(*Args*,_).
[2b] *vp'*([*Csem* | *Args*],*Sem*) :-
vp'(*Args*,*Sem*),
np(*Csem*).
[3b] *vp'*(*Args*,_) :-
Args=[*Ssem*],
np(*Ssem*).

(7) Finally, simplify the clauses and return to the standard form by removing unification goals. Remove superfluous arguments in literals. The result are the clauses [1'] to [3'] above.

CONCLUSIONS

We described a general method for inversion of logic grammars that transforms a parser into an efficient generator using an off-line compilation process that manipulates parser's clauses. The resulting "inverted-parser" generator behaves as if it was "parsing" a structured representation translating it into a well-formed linguistic string. The augmented grammar compilation procedure presented here is already quite general: it appears to subsume both the static compilation procedure of Strzalkowski (1990c), and the head-driven grammar evaluation technique of Shieber et al. (1990).

The process of grammar inversion is logically divided into two stages: (a) computing the collections of minimal sets of essential arguments (*MSEA*s) in predicates, and (b) rearranging the order of goals in the grammar so that at least one active *MSEA* is "in" in every literal when its expansion is attempted. The first stage also includes computing the "in" and "out" arguments. In the second stage, the goal inversion process is initialized by the procedure INVERSE, which recursively reorders goals on the right-hand sides of clauses to meet the *MSEA*-binding requirement. Deadlocked clauses which cannot be ordered with INVERSE are passed for the interclausal ordering with the procedure INTERCLAUSAL. Special treatment is provided for recursive goals defined with respect to properly ordered series of arguments. Whenever necessary, the direction of recursion is inverted allowing for "backward" computation of these goals. This provision eliminates an additional step of grammar normalization.

In this paper we described the main principles of grammar inversion and discussed some of the central procedures, but we have mostly abstracted from implementation level considerations. A substantial part of the grammar inversion procedure has been implemented, including the computation of minimal sets of essential arguments, and is used in a Japanese-English machine translation system.[24]

ACKNOWLEDGEMENTS

This paper is based upon work supported by the Defense Advanced Research Project Agency under Contract N00014-90-J-1851 from the Office of Naval Research, and by the National Science Foundation under Grant IRI-89-02304. Thanks to Marc Dymetman, Patrick Saint-Dizier, and Gertjan van Noord for their comments on an earlier version of this paper.

REFERENCES

Calder, Jonathan, Mike Reape and Henk Zeevat. 1989. "An Algorithm for Generation in Unification Categorial Grammar." *Proc. 4th Conf. of the European Chapter of the ACL*, Manchester, England, April 1989. pp. 233-240.

Colmerauer, Alain. 1982. *PROLOG II: Manuel de reference et modele theorique.* Groupe

[24] Further details can be found in (Peng and Strzalkowski, 1990; Strzalkowski and Peng, 1990; and Peng, forthcoming).

d'Intelligence Artificielle, Faculte de Sciences de Luminy, Marseille.

Dymetman, Marc and Pierre Isabelle. 1988. "Reversible Logic Grammars for Machine Translation." *Proc. 2nd Int. Conf. on Machine Translation*, Carnegie-Mellon Univ.

Dymetman, Marc, Pierre Isabelle and Francois Perrault. 1990. "A Symmetrical Approach to Parsing and Generation." *COLING-90*, Helsinki, Finland, August 1990. Vol. 3, pp. 90-96.

Estival, Dominique. 1990. "Generating French with a Reversible Unification Grammar." *COLING-90*, Helsinki, Finland, August 1990. Vol. 2, pp. 106-111.

Gardent, Claire and Agnes Plainfosse. 1990 "Generating from Deep Structure." *COLING-90*, Helsinki, Finland, August 1990. Vol 2, pp. 127-132.

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

Hasida, Koiti, Syun Isizaki. 1987. "Dependency Propagation: A Unified Theory of Sentence Comprehension and Generation." *IJCAI-87*, Milano, Italy, August 1987. pp. 664-670.

Ishizaki, Masato. 1990. "A Bottom-up Generation for Principle-based Grammars Using Constraint Propagation." *COLING-90*, Helsinki, Finland, August 1990. Vol 2, pp. 188-193.

Kay, Martin. 1984. "Functional Unification Grammar: A Formalism for Machine Translation." *COLING-84*, Stanford, CA, July 1984, pp. 75-78.

Landsbergen, Jan. 1987. "Montague Grammar and Machine Translation." Eindhoven, Holland: Philips Research M.S. 14.026.

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

Newman, P. 1990. "Towards Convenient Bi-Directional Grammar Formalisms." *COLING-90*, Helsinki, Finland, August 1990. Vol. 2, pp. 294-298.

Peng, Ping. forthcoming. "A Japanese/English Reversible Machine Translation System With Sublanguage Approach." Courant Institute of Mathematical Sciences, New York University.

Peng, Ping and Tomek Strzalkowski. 1990. "An Implementation of a Reversible Grammar." *Proc. 8th Canadian Conf. on Artificial Intelligence*, Ottawa, Canada, June 1990. pp. 121-127.

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

Shieber, Stuart, M. 1988. "A uniform architecture for parsing and generation." *COLING-88*, Budapest, Hungary, August 1988, pp. 614-619.

Shieber, Stuart, M., Gertjan van Noord, Robert C. Moore, Fernando C. N. Pereira. 1990. "A Semantic-Head-Driven Generation." *Computational Linguistics*, 16(1), pp. 30-42. MIT Press.

Steedman, Mark. 1987. "Combinatory Grammars and Parasitic Gaps." *Natural Language and Linguistic Theory*, 5, pp. 403-439.

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. 1990a. "An algorithm for inverting a unification grammar into an efficient unification generator." *Applied Mathematics Letters*, 3(1), pp. 93-96. Pergamon Press.

Strzalkowski, Tomek. 1990b. "How to Invert a Parser into an Efficient Generator: an algorithm for logic grammars." *COLING-90*, Helsinki, Finland, August 1990. Vol. 2, pp. 347-352.

Strzalkowski, Tomek. 1990c. "Reversible logic grammars for natural language parsing and generation." *Computational Intelligence*, 6(3), pp. 145-171. NRC Canada.

Strzalkowski, Tomek and Ping Peng. 1990. "Automated Inversion of Logic Grammars for Generation." *Proc. of 28th ACL*, Pittsburgh, PA, June 1990. pp. 212-219.

van Noord, Gertjan. 1990. "Reversible Unification Based Machine Translation." *COLING-90*, Helsinki, Finland, August 1990. Vol. 2, pp. 299-304.

Wedekind, Jurgen. 1988. "Generation as structure driven derivation." *COLING-88*, Budapest, Hungary, August 1988, pp. 732-737.