DIRECT AUTOMATED INVERSION OF LOGIC GRAMMARS *

Guido Minnen, Dale Gerdemann

Sonderforschungsbereich 340
University of Tübingen
Kleine Wilhelmstr. 113
72074-Tübingen
Federal Republic of Germany
E-mail: minnen@.sns.neuphilologie.uni-tuebingen.de

ABSTRACT

Reversibility of logic grammars in natural language processing is desirable for both theoretical and practical reasons. This report addresses this topic in describing a new approach to automated inversion of logic grammars: the Direct Inversion approach (DIA). A logic grammar is inverted by automatically altering the order of literals in the grammar and reformulating certain recursive procedures at compile time. The inversion process results in a new executable grammar, which is evaluated top-down and left-to-right (using a standard Prolog interpreter), but not left-to-right with respect to the original grammar. The DIA improves upon related approaches not only in being fully automated and computationally tractable, but also with respect to the class of grammars it is able to invert and the performance of the new executable grammar produced.

Keywords: reversible grammar, direction of computation, bidirectionality between parsing and generation.

1 INTRODUCTION

As noted by a number of researchers, reversibility of logic grammars in natural language processing is desirable for both theoretical and practical reasons ([12, 5, 6]). This report addresses

*This report was presented as a paper at the Fourth International Workshop on Natural Language Understanding and Logic Programming, September 29-October 1, 1993, Nara, Japan. The presented research was sponsored by Teilprojekt B4 "Constraints on Grammar for Efficient Generation" of the Sonderforschungsbereich 340 of the Deutsche Forschungsgemeinschaft. The authors wish to thank Erhard Hinrichs for stimulating discussions and advice and Dieter Martini for his programming contribution. Of course, the authors are responsible for all remaining errors.

this topic in describing a new approach to automated inversion of logic grammars: the Direct Inversion Approach (DIA). The DIA inverts a logic grammar by automatically altering the order of literals in the grammar and reformulating certain recursive procedures at compile time. The inversion process results in a new executable grammar, which is evaluated top-down and left-to-right (using a standard Prolog interpreter), but not left-to-right with respect to the original grammar.

The DIA closely relates to the Essential Arguments Approach (EAA), as reported upon by Strzalkowski ([14, 15, 16]). The EAA inverts a logic grammar in a two-step process. First, virtually all possible orderings of the literals in the grammar are investigated in a bottom-up fashion, through a precalculation of their *minimal sets of essential arguments* (MSEAs). Subsequently, the optimal evaluation order is achieved by ensuring that only those literals are called which are *ready* at a given moment, i.e. those that have at least one of their MSEAs instantiated.

The precalculation of MSEAs renders the EAA a generate-and-test approach which is computationally infeasible even for an off-line procedure. In the DIA the precalculation of MSEAs is circumvented, thereby yielding a direct, goal-directed and as a result of that computationally feasible approach to the automated inversion of logic grammars.

The DIA improves upon the EAA not only in being tractable, but also with respect to the class of grammars it is able to invert. Due to the goal-directedness of the DIA certain recursive procedures, that are rejected by the EAA as irreversible can be inverted successfully. Furthermore, for a specific class of grammars the new executable grammar produced by the DIA is more efficient.

In that the DIA achieves the optimal traversal order for the Prolog evaluation strategy in a direct or one-step, top-down reordering process, it closely resembles the static literal reordering approaches proposed in [2, 13]. However, in contrast to the DIA neither of these two approaches accomplishes this reordering automatically. Another related idea is the use of *magic sets* as in [1]. However, magic sets presuppose bottom-up execution of the transformed program, where the DIA uses top-down execution.

In order to facilitate the verification of the above made claims with respect to improvement upon the EAA and because a number of proposals made in the context of the EAA have been adapted and incorporated in the DIA, in the remainder of the report the DIA is contrasted with the EAA.

The set-up of the report is as follows. In section 2 the main ideas underlying EAA are described. Section 3 constitutes an elaborate description of the main inversion procedure of the DIA. The treatment of recursive clauses in the DIA (and the EAA) and its incorporation in the main inversion procedure are the subject of section 4. Section 5 contains concluding remarks and indicates directions for future research. The appendices contain the Prolog-code of the implementation of the DIA.

2 ESSENTIAL ARGUMENTS APPROACH

First, an informal explanation is given of how the EEA finds a new traversal order for the body of a clause in a grammar. Then the precalculation of the MSEAs of literals in a grammar and its relation to the actual reordering process accomplished by the EAA are described in more detail.

2.1 Informal Explanation

For the illustration of the reordering of the rhs-literals in a clause use is made of the simple extract from a grammar defining a wh-question (see next page)[1]. (The precalculated sets of MSEAs of the literals in the clauses are given between angle brackets[2].)

```
(1a) wh_question(P0,P,WhSem):-                     % <{{1=P0},{3=WhSem}}>
(1b)    wh_subject(P0,P1,Num,WhSubj),              % <{{1=P0},{4=WhSubj}}>
(1c)    wh_predicate(P1,P2,Num,
                 Tense,[WhSubj,WhObj],WhSem),      % <{{1=P1},{6=WhSem}}>

(1d)    wh_object(P2,P,WhObj).                     % <{{1=P2},{3=WhObj}}>

(2)  wh_subject([who|P],P,_,who).                  % <{{1},{4}}>
(3)  wh_subject([what|P],P,_,what).                % <{{1},{4}}>
     ...........................
(4)  wh_predicate([wrote|P],P,sing,
                 perf,[Subj,Obj],wrote(Subj,Obj)). % <{{1},{6}}>
(5)  wh_predicate([moved|P],P,sing,
                 perf,[Subj,Obj],moved(Subj,Obj)). % <{{1},{6}}>
     ...........................
(6)  wh_object([this|P],P,this).                   % <{{1},{3}}>
     ...........................
```

This grammar extract is parser-oriented in the sense, that the ordering of the literals of the body of clause (1) is such that all literals can be executed successfully, in case of parsing, i.e. the argument `P0` in the head-literal of clause (1) contains the input-string (e.g. `[who,wrote,this|P]`). However, in generation mode not `P0`, but `WhSem`, the argument carrying the encoding of the semantics of the expression (e.g. `wrote(who,this)`), is instantiated. Given the current ordering of the literals in (1) this results in an unconstrained call to `wh_subject`. Unconstrained calls to literals can easily lead to nontermination in the case of a recursive definition. Furthermore, in a more extensive grammar there may be many, possibly hundreds of unit clauses, which differ only on the values of their arguments. An unconstrained call to such unit clauses leads to massive nondeterminacy causing multiple backtracking, possibly stretching between distant points in the AND/OR computation tree.

By means of reordering the rhs-literals of (1) on the basis of their MSEAs, the EAA avoids calls which are not sufficiently instantiated. MSEAs indicate which arguments of a literal have to be instantiated to ensure successful execution. For example, literal (1c) in the grammar extract has two disjoint MSEAs: {`P1`} and {`WhSem`}. If one of these MSEAs is instantiated the literal can be executed successfully: in case `P1` is instantiated, (1c) unifies with either (4) or (5) (or it fails), just as in case only `WhSem` is instantiated.

In clause (1), given the information that `WhSem` is instantiated, (1c) is immediately identified as the rhs-literal to be executed first, because it has an instantiated MSEA ({`WhSem`}). The literals (1b) and (1d) can not be called first, as neither of them has an MSEA instantiated. Upon execution of (1c) the variables `Num`, `Tense`, `WhSubj` and `WhObj` are bound. Given this new binding information, one of the remaining literals (1b) and (1d), has to be selected. It is clear

[1] The grammar extract used in the example is also discussed in [7]).

[2] The MSEAs are represented either by position or position and variable name, whichever is more convenient.

that both can be executed safely now, because they have an instantiated MSEA ({`WhSubj`} and {`WhObj`}, respectively). Given the specification that `WhSem` is instantiated in literal (1a) the reordering process therefore finds two possible traversal orders for the literals in clause (1): c < b < d and c < d < b, and produces a new executable grammar in which the body of (1) is changed accordingly.

2.2 MSEAs and Essential Arguments

The precalculation of MSEAs can be viewed as the propagation of binding restrictions imposed upon the literals in the grammar by the *essential arguments*, i.e. the arguments that have to be bound at the time of execution, according to a particular permutation of the grammar.

There are two sorts of essential arguments:

1. *speeders*: arguments which have to be instantiated for a clause to be executed without introducing too much nondeterminacy[3]; and

2. *blockers*: arguments which have to be instantiated in order to be able to ensure the termination of a recursive clause.

Calculating the MSEAs (of the head-literal) of a particular clause presupposes the previous recursive calculation of MSEAs of the rhs-literals of that clause. On the basis of the MSEAs of the rhs-literals it is calculated, which arguments of the head-literal of the clause have to be instantiated in order to ensure the instantiation of an MSEA for each of the rhs-literals. Given a clause:

p(X_1,..,X_n):-
  r_1($X_{1,1}$,..,X_{1,k_1}),
  ..,
  r_s($X_{s,1}$,..,X_{s,k_s}).

there are two ways in which the instantiation of an argument X_g of the head-literal p can lead to the instantiation of an argument X_h of a rhs-literal r_j:

1. directly, due to the fact that the instantiated argument X_g of head-literal p is identical to X_h in the rhs-literal r_j or contains the argument X_h in the rhs-literal r_j as a subterm.

   Consider the grammar extract in §2.1 . As a consequence of the instantiation of the argument `WhSem` in (1a), the argument `WhSem` in literal (1c) is instantiated.

2. indirectly, in case the instantiation of the argument X_g of head-literal p leads to the instantiation of an MSEA of a rhs-literal r_i preceding r_j and an argument of r_i that is identical to X_h in the rhs-literal r_j or contains the argument X_h in the rhs-literal r_j as a subterm becomes instantiated upon execution[4].

   An example of the indirect instantiation of an argument in a rhs-literal as a result of the instantiation of an argument of a head-literal can be found in clause (1) of the example as well. The argument `WhObj` in literal (1d) becomes instantiated indirectly as a result

[3]Strzalkowski does not make a terminological distinction between speeders and blockers. The essential arguments in the previous example are all speeders.

[4]The set of rules Stzalkowski introduced to determine whether an argument in a literal is bound at the time the literal is submitted for execution or after the computation is completed is discussed in the context of its use in the DIA in §3.2.2.

of the instantiation of the argument `WhSem` in the head-literal (1a), if it is executed after the execution of literal (1c),, because upon the execution of literal (1c) with the argument `WhSem` instantiated `WhObj` becomes instantiated.

As the binding status of an argument of a rhs-literal of a clause possibly depends on the preceding rhs-literals in the body of that clause, the calculation of its MSEAs necessitates the investigation of virtually all possible orderings of the rhs-literals of this clause and its defining clauses: only in case two rhs-literals do not share any variables can their ordering can be considered irrelevant. An MSEA of a literal can be considered as representing a particular permutation of all its defining clauses.

2.3 Inversion

The actual reordering procedure of the EAA starts with the goal of inverting the top-level predicate on the basis of the intended input-arguments and the precalculated set of all MSEAs of this predicate. This is done by making sure that every rhs-literal has an instantiated MSEA, i.e. it has a minimal set of essential arguments of which all elements are bound, at the time of execution. This necessitates the recursive inversion of the clauses defining these rhs-literals to make sure that the rhs-literals in the body of these clauses have an instantiated MSEA as well, and so on.

Once the inversion procedure has found an instantiated MSEA for all the rhs-literals of the top-level predicate (and therefore for all their defining clauses), it has found a permutation of the grammar that is compatible with the a priori known information, using the information deduced from the precalculation of all permutations, i.e. guided by the MSEAs information.

Due to its top-down nature the inversion procedure requires immediate access to the MSEAs of the rhs-literals of the clauses defining the top-level predicate, which given the bottom-up character of the precalculation of MSEAs, involves calculating the MSEAs of all predicates in the grammar. This renders the EAA a generate-and-test approach, which is computationally infeasible even for an off-line procedure. Strzalkowski notices that an integration of the precalculation of MSEAs and the reordering process is desirable. However the partial implementation of the EAA ([8, 17]) presupposes the precalculation of the MSEAs of all the literals in the grammar.

3 DIRECT INVERSION APPROACH

In the DIA the precalculation of MSEAs is circumvented: instead of propagating the binding restrictions imposed upon the literals in the grammar by the essential arguments, these binding restrictions are tested locally without identifying the essential arguments as such. This results in a direct, goal-directed and as a result of that computationally feasible approach to the automated inversion of logic grammars. It should be noted that the DIA, like the EAA, can operate only upon the declarative specification of a logic grammar and is not prepared to deal with extralogical control operators such as the cut.

First, by briefly returning to the simple grammar extract in §2.1, it is informally explained how the literals in the body of a clause are reordered without the MSEAs of the literals to guide the inversion process. As in the previous section the focus of attention is on the efficient execution of the grammar. The treatment of recursive clauses (in the DIA, as well as in the EAA) and its incorporation in the overall inversion process sketched below is discussed in section

4. Second, a more detailed description of the inversion process of the DIA is given, followed by an elaborate example.

3.1 Inversion without MSEAs

The DIA inverts a grammar solely on the basis of the intended input-arguments. The body of clause (1) of the grammar extract in §2.1 is again reordered according to the information that the third argument (`WhSem`) of literal (1a) is instantiated.

```
(1a) wh_question(P0,P,WhSem):-
(1b)     wh_subject(P0,P1,Num,WhSubj),
(1c)     wh_predicate(P1,P2,Num,Tense,[WhSubj,WhObj],WhSem),
(1d)     wh_object(P2,P,WhObj).
```

The process of reordering the body of a clause is illustrated as shifting the literals of the body of this clause (`OLdRhs`) one by one to the new reordered body of this clause: `NewRhs`. The situation when the reordering process of clause (1) starts can be represented as:

```
OldRhs:  (1b) wh_subject(P0,P1,Num,WhSubj)
         (1c) wh_predicate(P1,P2,Num,Tense,[WhSubj,WhObj],WhSem)
         (1d) wh_object(P2,P,WhObj)

NewRhs:  -
```

The only literal in `OldRhs` which has an instantiated argument as a result of the instantiation of the argument `WhSem` in (1a) is literal (1c) (`WhSem`). The fact that (1c) is the literal with the largest number of instantiated arguments leads to its selection as the first literal to be executed. Subsequently, the clauses defining literal (1c) are investigated to determine whether (1c) can be executed without introducing too much nondeterminacy given the binding of its arguments. This is done by computing the maximal number of clauses with which (1c) unifies given the argument `WhSem` instantiated.

```
(4) wh_predicate([wrote|P],P,sing,perf,[Subj,Obj],wrote(Subj,Obj)).
(5) wh_predicate([moved|P],P,sing,perf,[Subj,Obj],moved(Subj,Obj)).
```

Given the argument `WhSem` instantiated, literal (1c) does not introduce any nondeterminacy: the maximal number of clauses with which literal (1c) unifies is one. Whatever the value of the argument `WhSem`, (1c) unifies either with only one of the defining unit clauses (4) and (5) or fails. Therefore the selection of (1c) is approved of and it is shifted to `NewRhs`. The new situation looks as follows:

```
OldRhs:  (1b) wh_subject(P0,P1,Num,WhSubj)
         (1d) wh_object(P2,P,WhObj)

NewRhs:  (1c) wh_predicate(P1,P2,Num,Tense,[WhSubj,WhObj],WhSem)
```

If the maximal number of clauses with which literal (1c) unifies would have been larger then one, its selection could have been rejected depending on the maximal degree of nondeterminacy the inverted grammar is allowed to introduce. In case of the rejection of the selection of a literal another rhs-literal would be selected by means of backtracking. Note that only after its defining clauses have been investigated, it is possible to determine locally what degree of nondeterminacy a call to a literal introduces.

If a literal is defined by one or more nonunit clauses, then these clauses have to be inverted recursively in accordance with the information concerning the instantiation of its arguments. The clauses defining the rhs-literal in clause (1) are all unit clauses and therefore the selection of a literal does not trigger the recursive inversion of any other clauses.

Now it is computed which arguments of the literals in `OldRhs` become instantiated upon the execution of the literals in `NewRhs`. `NewRhs` only contains literal (1c) upon which execution the arguments `Num`, `Tense`, `WhSubj` and `WhObj` become instantiated. This information is used to select the next rhs-literal that is to be shifted from `OldRhs` to `NewRhs`. Of the literals remaining in the `OldRhs` literal (1b) has the largest number of instantiated arguments and therefore it is selected as the literal to be executed next. Given the argument `WhSubj` instantiated the execution of literal (1b) does not introduce any nondeterminacy and therefore after its defining clause is considered its selection is determined suitable to be executed next:

```
OldRhs:  (1d) wh_object(P2,P,WhObj)

NewRhs:  (1c) wh_predicate(P1,P2,Num,Tense,[WhSubj,WhObj],WhSem)
         (1b) wh_subject(P0,P1,Num,WhSubj)
```

Now there is only one rhs-literal left in `OldRhs`: (1d). It is still necessary to compute which arguments of the literals in `OldRhs` become instantiated upon execution of the literals in `NewRhs`, in order to determine which arguments of literal (1d) are instantiated at the time it is executed. There are no new instantiations in `OldRhs` resulting from the execution of literal (1b) contributing to the instantiation of the variables of literal (1d) (only its third argument `WhObj` is instantiated as a result of the execution of literal (1c)), but this suffices for its execution without introducing too much nondeterminacy.

After (1d) is shifted to `NewRhs` there are no rhs-literals left in `OldRhs` and therefore it is known that a new ordering of the body of (1) is found, i.e. the clause is inverted successfully.

```
OldRhs:  -

NewRhs:  (1c) wh_predicate(P1,P2,Num,Tense,[WhSubj,WhObj],WhSem)
         (1b) wh_subject(P0,P1,Num,WhSubj)
         (1d) wh_object(P2,P,WhObj)
```

It is important to notice that the DIA does not produce two equivalent new orderings of the body of clause (1), as the EAA, but only one: $c < b < d$. The ordering where (1b) precedes (1d) is preferred, because at the time of its execution it has a larger number of instantiated arguments.

3.2 Direct Inversion

3.2.1 Main Inversion Procedure

A more detailed description of the DIA is given on the basis of the main inversion predicates of the Prolog implementation of the DIA. For expository reasons the predicates have been simplified. The actual code can be found in appendix B.

The inversion process starts off with a call to the predicate `inverse/3` with the top-goal (`Literal`), i.e. the call for the top-level predicate of the input-grammar, the intended input-arguments (`BoundArgs`) and a specification of the maximal degree of nondeterminacy (`Degree`) the inverted grammar is allowed to introduce (see also §3.2.4 and §3.3.3):

```
inverse(Literal,BoundArgs,Degree):-
   defining_clauses(Literal,DefClauses),
   degree_of_nondeterminacy(DefClauses,Degree),
   inverse_procedure(DefClauses,NewDefClauses,BoundArgs,Degree),
   result(NewDefClauses).
```

`inverse/3` retrieves the clauses defining the top-goal (`DefClauses`) from the input-grammar, checks whether the degree of nondeterminacy introduced by the defining clauses does not exceed the specified maximal degree of nondeterminacy and calls `inverse_procedure/4` with `DefClauses` and `BoundArgs`, and the context argument `Degree`.

```
inverse_procedure([],[],_,_):-
inverse_procedure([[Head|OldRhs]|DefClauses],[[Head|NewRhs]|NewDefClauses],
                  BoundArgs,Degree):-
   inverse_clause(Head,OldRhs,NewRhs,BoundArgs,Degree),
   inverse_procedure(DefClauses,NewDefClauses,BoundArgs,Degree).
```

For each of the clauses in `DefClauses`, `inverse_procedure/4` calls the predicate `inverse_clause/5` which takes care of the inversion of an individual clause. Upon the execution of `inverse_clause/5` the output-argument `NewRhs` contains the reordered body of the defining clause `[Head|OldRhs]`.

```
inverse_clause(Head,[],NewRhs,_,_).
inverse_clause(Head,OldRhs,NewRhs,BoundArgs,Degree):-
   bound_args_oldrhs(OldRhs,BoundArgs),
   select_rhs_literal(SelectedLiteral,OldRhs,OldRhsSub),
   inverse(SelectedLiteral,BoundArgsSelectedLiteral,Degree),
   append(NewRhs,SelectedLiteral,NewestRhs),
   bound_args_newrhs(Head,NewestRhs,OldRhsSub,NewBoundArgs),
   inverse_clause(Head,NPred,OldRhsSub,NewestRhs,NewBoundArgs,Degree).
```

On the basis of the intended input-arguments (`BoundArgs`), `bound_args_oldrhs/2` computes which arguments in `OldRhs` are instantiated. Subsequently, `select_rhs_literal/3` selects the literal with the largest number of instantiated arguments (`SelectedLiteral`). For the motivation for the use of this heuristic rule to select a rhs-literal see §3.2.3. Upon the selection of a rhs-literal `inverse/3` is called recursively for this literal to ensure that the binding of the arguments of `SelectedLiteral` allow a successful inversion of its defining clauses.

In case the recursive call to `inverse/3` is successful, the selected rhs-literal is moved to `NewestRhs`. Subsequently, the arguments which get instantiated upon execution of the literals in `NewestRhs` are calculated by `bound_args_newrhs/4` (`NewBoundArgs`). According to which rules the predicate `bound_args_newrhs/4` determines the binding status of an argument in a literal is the subject of §3.2.2. Now `inverse_clause/5` is called again to select the next rhs-literal. When all rhs-literals are shifted from `OldRhs` to `NewestRhs`, the optimal traversal order for the body of the defining clause is achieved.

When `inverse_procedure/4` has called `inverse_clause/5` for all clauses defining a literal, the resulting new inverted procedure (`NewDefClauses`) is stored by `result/1`. As a result of the fact that `inverse_clause/5` calls `inverse/3` recursively for all the rhs-literals in the body of a clause it inverts, a successful call to `inverse/3` for the top-goal means that the input-grammar is inverted and due to the calls to `result/1` at each level of the recursion the new executable grammar is stored.

3.2.2 Binding Status of Arguments

To determine whether an argument in a literal is bound at the time the literal is submitted for execution or after the computation is completed, the predicate `bound_args_newrhs/4` make use of a set of rules proposed by Strzalkowski ([15]). On the basis of the instantiation of the intended input-arguments of the top-goal an argument in a literal in the input-grammar gets assigned a binding status:

- if an argument in a literal is instantiated at the time the literal is submitted for execution, the rules assign it the status *in*.
- If an argument in a literal is instantiated after the execution of the literal is completed, the rules assign it the status *out*.
- If it is unknown whether an argument in a recursively defined literal is instantiated upon the execution of the literal or not, the rules assign it the status *unknown*.

The rules according to which the status of an argument in a literal is determined look as follows:

An argument X of the literal *pred(..X..)* on the right-hand side of a clause is *in* if:
(a-in) it is a constant;
(b-in) it is a function and all its arguments are *in*;
(c-in) it is *in* or *out* in some previous literal on the right-hand side of the same clause; or
(d-in) it is *in* in the head literal on the left-hand side of the same clause.

An argument X in the head literal L = *pred(..X..)* on the left-hand side of a clause is *in* if:
(a-in) it is a constant;
(b-in) it is a function and all its arguments are *in*;
(e-in) L is the top-goal and X is an intended input-argument;
(f-in) X occurs more than once in L and a least one of these occurences is *in*
(g-in) for every literal *L1* = *pred(..Y..)* unifiable with L on the right-hand side of any clause with the head predicate *pred1* different from *pred*, and such that Y unifies with X, Y is *in* in *L1*.

An argument X of the literal *pred(..X..)* on the right-hand side of a clause is *out* if:
(a-out) it is a functional expression and all its arguments are either *in* or *out* with at least one argument being *out*;
(b-out) for every clause with the head literal *pred(..Y..)* unifiable with *pred(..X..)* and such that Y unifies with X, Y is either *in*, *out* or *unknown* and Y is marked *in* or *out* in at least one case.

An argument X of the literal *pred(..X..)* on the lhs of a clause is *out* if:
(c-out) it is *out* in literal *pred1(..X..)* on the right-hand side of this clause, providing that *pred1*

$\neq$ *pred*; if pred1 = pred then X is marked *unknown*.

The implementation of this set of status-assignment rules can be found in appendix C. It is important to note that the set of *in*-arguments and the set of *out*-arguments in a grammar are not disjoint. However, the status computation implemented in the context of the DIA, does not assign arguments more than one status: once an argument is assigned a status, this status is not changed. As the rules to determine whether an argument is *in* are applied prior to the rules to determine whether an argument is *out*, arguments which are *in*, as well as *out* according to the rules get assigned the status *in*. Note that if an argument is *in* in some complex term, then the argument is *in* by itself.

in and *out* status-assignment is not exhaustive according to the proposed rules. In some cases the status of an argument is nondeterminate locally (within a clause). The third status *unknown* is used not to cover the non-*in* and non-*out* arguments, but to assign to certain arguments of recursive clauses of which it is unknown whether they are *out*, or not, without actually tracing the entire process to determine whether the recursion stops successfully. On the assumption that the termination of a call to a recursively defined literal is successful, this status-underspecification does not prohibit the correct *out* status-assignment to the arguments of a call to a literal which is defined by a recursive procedure[5], as is illustrated by the following example[6]:

```
(1a) pred1(...):-
(1b)    pred(P,Q).

(2a) pred(f(X),Y):-
(2b)    pred(X,Y).
(3)  pred(const1,const2).
```

Rule b-out correctly assigns the *out* status to `Q` in literal (1b), even though `Y` in the literal (2a) gets assigned the status *unknown* by rule c-out: the *in* status of the corresponding argument in literal (3) (the only other possible expansion of (1b)), the constant `const2`, suffices to determine that `Q` in literal (1b) is instantiated upon the execution of (1b) in case the recursion terminates successfully.

The status an argument in the input-grammar gets assigned according to the above rules is indicated by means of indices. Each argument in the input-grammar gets an index which remains uninstantiated or becomes instantiated to the value *in*, *out* or *unknown* (see also appendix A). The above clauses annotated with the status-indices look as follows:

```
(1a) pred1(...):-
(1b)    pred(P-out,Q-out).

(2a) pred(f(X)-unknown,Y-unknown):-
(2b)    pred(X-out,Y-out).
(3)  pred(const1-in,const2-in).
```

Strzalkowski claims that the above set of rules suffices to compute the binding status of arguments to support the inversion of a logic grammar, but his treatment of recursive clauses necessitates an extension of the set of rules to be able to identify not only arguments in a literal

[5] If the recursion does not stop successfully, it either fails or loops forever in which case the question of the *out* status of the arguments involved becomes irrelevant.

[6] This example is taken from [15].

which become fully instantiated upon the execution of this literal, but also arguments in a literal which are *partially out*, i.e. become ground only upon the execution of this literal and one or more of the other rhs-literals in the body of a clause ([15, 16]). In the DIA, the extended set of rules to assign a fourth status to arguments which are partially out, is not implemented, as for the treatment of recursive clauses in the DIA the additional information provided by the suggested extension of the rules is not used. This does not mean that the binding status computation can not be improved by means of an extension of the set of rules or a different algorithm to determine the dataflow in a logic grammar (see section 5).

3.2.3 Selection of a Rhs-literal

The predicate `select_rhs_literal/3` uses a heuristic rule for the selection of the literal (from the `OldRhs`) to be executed next. Instead of selecting a rhs-literal nondeterministically and subsequently checking whether the associated binding information, i.e. the arguments of the literal which are determined to be bound, allow sucessful execution, the number of wrong choices is reduced by always first selecting the rhs-literal with the largest number of instantiated arguments[7]. There are two reasons for using this heuristic rule. If the rhs-literal with the largest number of instantiated arguments is chosen then:

1. if the selected literal is suitable, the subsequent recursive calls to `inverse/3` for its defining clauses are minimally restricted. This is based on the observation that the larger the number of instantiated arguments in the head-literal of a clause is, the less restricted the reordering of its body is.

2. If two or more rhs-literals of a clause are equally suitable to be executed next, i.e. they can be executed successfully given the current binding information, selecting the rhs-literal with the largest number of instantiated arguments ensures the clause to fail as early as possible: if possible the literal with the largest number of bound arguments is fired prior to the other literals in the body of the same clause.

   Fail-as-early-as-possible is a useful goal-ordering strategy to increase the efficiency of the execution of a grammar ([11]), as it reduces the amount of computation to be performed to reach to the conclusion that a particular OR-branch in the computation-tree leads to a fail-node. The more arguments of a literal are bound at the time of execution, i.e. the more constrained the call to a literal is, the earlier a call to this literal fails. In the EAA attention is paid only to the *minimal* sets of essential arguments of a literal, thereby neglecting this optimization of the traversal order.

   An illustration of this optimization is the fact that given the grammar extract in §2.1 the DIA finds only one "optimal" traversal order for clause (1). The EAA finds two traversal orders which are considered equally good.

It is important to notice that the use of the heuristic rule for the selection of a rhs-literal affects the goal-directedness of the DIA only marginally.

[7]This heuristic rule could be specialized to give preference to the literal with the largest number of instantiated arguments relative to its total number of arguments. The current implementation selects literals on the basis of the absolute number of instantiated arguments.

3.2.4 Degree of Nondeterminacy

Checking the degree of nondeterminacy introduced by the call to a selected literal can be considered the DIA-equivalent of a speeder in the EAA. However, there are three important differences:

1. The degree of nondeterminacy introduced by calling a rhs-literal is checked only locally, i.e. at the moment the inversion process retrieves the clauses defining the literal. Recall that in the EAA the implications of essential arguments are percolated throughout the complete grammar.

2. Due to the checking of the degree of nondeterminacy at the time the literals become reordered, it is always known which arguments of a literal are bound, which makes the identification of speeders superfluous: it is only of interest whether the set of instantiated arguments "as a whole" allow the execution of a literal without the introduction of too much nondeterminacy (see below).

3. The degree of nondeterminacy introduced by calling a rhs-literal is checked not only in case the defining clauses are all unit clauses, but also in case the procedure contains one or more nonunit clauses. The EAA computes speeders only for the former.

Suppose the grammar extract in §2.1 contains a larger number of unit clauses defining `wh_predicate`, e.g.:

```
wh_predicate([wrote|P],P,sing,perf,[Subj,Obj],wrote(Subj,Obj)).
wh_predicate([wrote|P],P,plur,perf,[Subj,Obj],wrote(Subj,Obj)).
wh_predicate([moved|P],P,sing,perf,[Subj,Obj],moved(Subj,Obj)).
wh_predicate([moved|P],P,plur,perf,[Subj,Obj],moved(Subj,Obj)).
```

Suppose one of the literals remaining in `OldRhs` is the literal `wh_predicate` and its third (`Num`), fourth (`Tense`) and fifth (`WhSem`) argument is bound, then the literal is approved of to be executed next, because there is at most one defining clause with which it unifies. The fact that the binding of `Tense` does not add to the determinacy of the call to `wh_predicate` is completely ignored. How the instantiation of the individual arguments contributes to the successful execution of `wh_predicate` is irrelevant. In the EAA, the contribution of the instantiation of each individual argument is determined through the precalculation of the MSEAs, even in case it is never instantiated prior to execution. So, for example, the MSEAs for the literal `wh_predicate` in clause (1) in §2.1 are {`P1`} and {`Whsem`}. The information that in generation mode the argument `P1` is never instantiated prior to the execution of `wh_predicate` is not taken into account.

3.3 Chart Representation of Intermediate Results

The DIA investigates orderings of literals in the grammar by means of backtracking: in case a rhs-literal is selected by `select_rhs_literal/3` and it turns out that given the current binding information execution of the rhs-literal introduces too much nondeterminacy, backtracking leads to the selection of another rhs-literal. The backtracking strategy is extended by maintaining a chart of intermediate inversion results in order to eliminate redundant computation.

The chart is used not only to increase the efficiency of the inversion process, but also to keep the new executable grammar as small as possible by avoiding needless copying of predicates which are called more than once in the grammar. Furthermore, it is used to identify direct and indirect recursive clauses and therefore indispensible to ensure the termination of the inversion process.

The chart is represented by means of unit clauses which get asserted to the database during the inversion process. There are three kinds of dynamic predicates which get asserted to the chart during the inversion of an input-grammar:

1. `success(Indices,Procedure)`: success-entries in the chart are used to represent the successful inversion of a procedure `Procedure`[8]. Where `Indices` indicates which clauses instigated the inversion of the procedure.

2. `failure(Index,Literal,Degree)`: failure-entries in the chart are used to represent the previous failure of the inversion of the clauses defining `Literal`. Where `Degree` is the minimal degree of nondeterminacy introduced by the clauses defining `Literal` and `Index` refers to the clause that instigated the inversion of the clauses defining `Literal`.

3. `opened(PredicateName)`: opened-entries in the chart are used to detect directly and indirectly recursive procedures. In case of a call for the inversion of a procedure with predicate name `PredicateName` it is checked whether there is an opened-entry for this predicate name, as suggested in [15]. Upon the inversion of a procedure the corresponding opened-entry is retracted from the chart.

The advantage of the explicit storage of chart-entries is that more intermediate results can be reused. Moreover, in combination with the goal-directedness of the overall inversion process this leads to a significant improvement upon the EAA in case one tries to obtain the most efficient inversion of a grammar by means of consecutively incrementing the admissible degree of nondeterminacy (see §3.3.3).

Upon the successful inversion of the clauses defining the top-goal, the success-entries contain the internal representation of the new executable grammar. Because of the combination of a backtrack strategy and the maintenance of a chart representation of intermediate results, it is necessary to weed out success-entries in the chart representing inverted procedures not directly or indirectly called by the clauses defining the top-goal.

3.3.1 Avoidance of Unnecessary Multiple Inversion

By means of keeping track of which clauses have been inverted according to which binding information, needless copying of clauses in a grammar is avoided. Before `inverse_procedure/4` is called for the clauses defining a literal, it is checked in the chart whether these clauses have been inverted before and according to which binding information, to determine whether the result of the previous or the current inversion of these clauses can be reused. There are three possibilities:

1. The new procedure that resulted from a previous successful call to `inverse_procedure/4` for the clauses defining a literal can also be considered as suitable for the current call. This is the case, when the set of bound arguments, i.e the binding information, associated with the literal for the previous call is a proper subset of the set of bound arguments associated with the literal for the current call. (In which case the previous call is called more restrictive.) Or, when the binding information for both calls is identical.

[8] Note that due to the status-indices used to indicate the status of an argument, by storing a literal/procedure in the chart, by means of a failure/success-entry the associated binding information is stored as well.

For example, in case the clauses defining the literal `lit(A,B,C,D)` have been inverted successfully given the information that the arguments `A` and `B` are bound, then any call to `inverse_procedure/4` for the clauses defining `lit(A,B,C,D)` with at least the arguments `A` and `B` instantiated can be considered as redundant.

2. The new procedure that results from the current call to `inverse_procedure/4` for the clauses defining a literal can be considered as suitable also for a previous call. This is the case, when the binding information associated with the current call is identical or more restrictive than the previous call.

   For example, in case the clauses defining the literal `lit(A,B,C,D)` have been inverted successfully given the information that the arguments `A` and `B` are bound, then the result from a call with only argument `A` instantiated can substitute the clauses resulting from the previous call to `inverse_procedure/4`.

3. Neither the current, nor (any of) the previous calls to `inverse_procedure/4` for the clauses defining a literal can be considered as reusable. This is the case when the binding information associated with the calls is not comparable, i.e. none of the calls is identical to or less or more restrictive than another call.

   For example, in case the clauses defining the literal `lit(A,B,C,D)` have been inverted successfully given the information that the argument `A` is bound, a new call to `inverse_procedure/4` for these clauses with the information that the argument `B` is bound is not comparable to the previous inversion: the current binding of the arguments is neither less or more restrictive, nor identical to the previous binding. This necessitates the keeping of the results of both calls to `inverse_procedure/4`, by means of assigning a new predicate name to the new resulting clauses and the corresponding literal.

3.3.2 Fail Anticipation

The fact that the chart is also used to store information concerning calls to `inverse/3` which have failed previously allows the avoidance of redundant computations by means of *fail anticipation*. For example, if the chart contains the information that the reordering of the clauses defining rhs-literal `lit(A,B,C,D)` of clause `Index` with arguments `A` and `B` bound led to failure and the minimal degree of nondeterminacy it introduces is 6:

```
failure(Index,lit(A-in,B-in,C,D),6),
```

it is known in advance that any call to `inverse/3` for `lit(A,B,C,D)` with identical or more restrictive binding information given an admissible degree of nondetermincay smaller than 6 will lead to failure as well.

The minimal degree of nondeterminacy is used in combination with consecutively incrementing the degree of nondeterminacy the new executable grammar is allowed to have. It is determined by means of the success- and failure-entries for the (clauses defining the) rhs-literals in the body of its defining clauses. In case there exists a rhs-literal of a clause for which a failure-entry is found in the chart, but no success-entry, it is known that the admissible degree of nondeterminacy should at least be increased up to the minimal degree of nondeterminacy stored in this failure-entry, otherwise the inversion of the clause will fail again. It is possible that there exists more than one of such failure-entries in which case the smallest degree is stored in the new failure-entry.

3.3.3 Maximal Degree of Nondeterminacy

An important advantage of the use of the chart in combination with the goal-directedness of the DIA becomes apparent, when it is impossible to reorder the literals in the grammar given a particular degree of nondeterminacy. In that case one wants to relax the admissible degree of nondeterminacy, such that the most efficient inversion of the grammar is produced.

Because it is possible to reuse the intermediate results of previous attempts to invert the grammar, the inversion of a grammar according to a less constraining degree of nondeterminacy is very efficient. Only the clauses for which no inversion has been found in the previous attempts need to be reconsidered. This is implemented through an iterative deepening technique ([10]) of consecutively incrementing the admissible degree of nondeterminacy, until a specified maximal degree of nondeterminacy is reached:

```
determinacy_loop(TopGoal,BoundArgs,Degree,MaxDegree):-
   ( inverse(TopGoal,BoundArgs,Degree)
   ; (Degree < MaxDegree ->
     NewDegree is Degree + 1,
     determinacy_loop(TopGoal,BoundArgs,NewDegree,MaxDegree))
   ).
```

Thus instead of calling `inverse/3` with a specific degree of nondeterminacy of, for example, 4, `determinacy_loop/4` is called with `Degree = 1` and `MaxDegree = 4`. Given this input-specification `determinacy_loop/4` calls `inverse/3` at first with `Degree = 1` and as long as no complete inversion of the input-grammar is found it consecutively increments `Degree` up to and including 4.

As a result of fail anticipation through the storage of the minimal degree of nondeterminacy introduced by the call to a literal in the failure-entries in the chart, it is possible to determine which `Degree` will never lead to the inversion of the input-grammar: in case the failure-entry for the clauses defining the topgoal has a minimal degree of nondeterminacy which is larger than the admissible degree of nondeterminacy, it is known in advance that the inversion of the input-grammar according to this `Degree` will never lead to success[9].

The flexibility of the DIA with respect to the admissible degree of nondeterminacy constitutes a significant advantage over the EAA, because the EAA has to recalculate the MSEAs for all the literals in the grammar every time the admissible degree of nondeterminacy changes. This means that in order to find the most efficient inversion of a grammar which introduces a degree of nondeterminacy of 4, it is necessary to investigate virtually all permutations of the literals in the grammar four times. It is possible to set the admissible degree of nondeterminacy to 4 directly, but this can possibly lead to a less efficient grammar, because then the whole grammar is inverted to this degree of nondeterminacy. Moreover, there is no way to determine the degree of nondeterminacy of a grammar in advance.

3.4 Example

The DIA is illustrated through the direct inversion of a sample (parser-oriented) grammar fragment. Consider the following Prolog-coded unification grammar for YN-questions[10]:

[9] This usage of the minimal degree of nondeterminacy in failure-entries resembles the use of counters to constraint an iterative deepening technique as in [10].

[10] The grammar used in the example is also discussed in [7].

```
(1a)  sentence(P0,P,ques(askif(S))):-
(1b)     yesnoq(P0,P,askif(S)).

(2a)  yesnoq(P0,P,askif(S)):-
(2b)     aux_verb(P0,P1,Num,Pers,Form,Aux),
(2c)     subj(P1,P2,Num,Pers,Subj),
(2d)     main_verb(P2,P3,Form,[Subj,Obj],Verb),
(2e)     obj(P3,P4,_,_,Obj),
(2f)     adv(P4,P,[Verb],S).

(3)   aux_verb([have|P],P,sing,one,pres_perf,have(pres_perf,sing)).
      ...........................
(4)   aux_verb([am|P],P,sing,one,pres_cont,be(pres_cont,sing-1)).
      ...........................
(5)   aux_verb([do|P],P,sing,one,pres,do(pres,sing-1)).
(6)   aux_verb([do|P],P,sing,two,pres,do(pres,sing-2)).
(7)   aux_verb([does|P],P,sing,three,pres,do(pres,sing-3)).
(8)   aux_verb([do|P],P,plur,one,pres,do(pres,pl-1)).
      ...........................
(9a)  subj(P0,P,Num,Pers,Subj):-
(9b)     np((P0,P,Num,Pers,su,Subj).

(10a) obj(P0,P,Num,Pers,Obj:-
(10b)    np((P0,P,Num,Pers,ob,Obj).

(11a) np(P0,P,Num,Pers,Case,NP):-
(11b)    noun((P0,P,Num,Pers,Case,NP).
(12a) np(P0,P,Num,Pers,Case,NP):-
(12b)    pnoun((P0,P,Num,Pers,Case,NP).

(13)  pnoun([you|P],P,sing,two,su,you).
(14)  pnoun([him|P],P,sing,three,ob,him).
      ...........................
(15)  main_verb([see|P],P,pres,[Subj,Obj],see(Subj,Obj).
(16)  main_verb([seen|P],P,pres_perf,[Subj,Obj],seen(Subj,Obj).
(17)  main_verb([saw|P],P,perf,[Subj,Obj],saw(Subj,Obj).
      ...........................
(18)  adv([often|P],P,[Verb],often(Verb)).
(19)  adv([seldom|P],P,[Verb],seldom(Verb)).
      ...........................
```

For this grammar fragment to be used in generation mode successfully, clause (2) has to be reordered, therefore the focus of attention is on this clause. Initially, `determinacy_loop/4` is called with the call to clause (1), i.e. the top-goal, and the information that its third argument `ques(askif(S))` is fully instantiated, i.e. is the intended input-argument (stored in `BoundArgs`) and a specification of a maximal admissible degree of nondeterminacy of, for example, 3. This immediately leads to a call to `inverse/3` with the same input-arguments except for the admissible degree of nondeterminacy which is set to 1. The clauses defining the top-goal are now retrieved from the input-grammar and administrated as opened for inversion. Whatever the intended input-arguments are no nondeterminacy is introduced, because clause (1) is the only clause defining the top-goal. `bound_args_oldrhs/2` computes the implications of the instantiation of the third argument of the head-literal: `askif(S)` in (1b) is instantiated. Being the only literal in the body of clause (1), (1b) is selected which leads to a call to `inverse_procedure/4` with clause (2), the clause defining literal (1b), and a call to `inverse_clause/5` with clause (2). Again the fact that these clauses are in the process of being inverted is stored in the chart by means of an opened-entry. Adopting the same way of representing the stages of the inversion process of this clause as in §3.1, the situation looks as follows now[11]:

```
OldRhs:  (2b) aux_verb(P0,P1,Num,Pers,Form,Aux)
         (2c) subj(P1,P2,Num,Pers,Subj)
         (2d) main_verb(P3,P4,Form,[Subj,Obj],Verb)
         (2e) obj(P4,P5,_,_,Obj)
         (2f) adv(P5,P,[Verb],S-in)

NewRhs:  -

Chart:
(1)   opened(sentence)
(2)   opened(yesnoq)
```

`bound_args_oldrhs/2` determines that the only literal in `OldRhs` with an instantiated argument is literal (2f) and (2f) is therefore selected by `select_rhs_literal/3` and `inverse/3` is called recursively. Given the instantiation of its fourth argument a call to (2f) introduces no nondeterminacy, because it either unifies with (18) or (19), or it fails, and (2f) is shifted to `NewRhs` and a success-entry is added to the chart:

```
OldRhs:  (2b) aux_verb(P0,P1,Num,Pers,Form,Aux)
         (2c) subj(P1,P2,Num,Pers,Subj)
         (2d) main_verb(P3,P4,Form,[Subj,Obj],Verb-in)
         (2e) obj(P4,P5,_,_,Obj)

NewRhs:  (2f) adv(P5,P,[Verb]-out,S-in)

Chart:
(1)   opened(sentence)
(2)   opened(yesnoq)
(3)   success([2],adv([often|P],P,[Verb],often(Verb)-in),...)
```

Upon execution of (2f), the main verb semantics (`Verb`) becomes instantiated and as a result of that gets stored in `NewBoundArgs`. Again, there is only one rhs-literal with an instantiated argument: using the heuristic rule, (2d) is selected and as it meets the determinacy requirements the new situation becomes:

[11] Only in case the status-index associated with an argument is instantiated it is given.

```
OldRhs:  (2b) aux_verb(P0,P1,Num,Pers,Form-in,Aux)
         (2c) subj(P1,P2,Num,Pers,Subj-in)
         (2e) obj(P4,P5,_,_,Obj-in)

NewRhs:  (2f) adv(P5,P,[Verb]-out,S-in)
         (2d) main_verb(P3,P4,Form-out,[Subj,Obj]-out,Verb-in)

Chart:
(1)   opened(sentence)
(2)   opened(yesnoq)
(3)   success([2],adv([often|P],P,[Verb],often(Verb)-in),...)
(4)   success([2],main_verb([see|P],P,pres,[Subj,Obj],see(Subj,Obj)-in),...)
```

`bound_args_newrhs/4` and `bound_args_oldrhs/3` are called to determine which arguments in `OldRhs` become instantiated upon the execution of the literals in `NewRhs`: `Subj`, `Obj` and `Form`.

Now there are three rhs-literals with one instantiated argument. `select_rhs_literal/4` selects one of them nondeterministically (the heuristic rule does not give a preference): `aux_verb`. However literal (2b) can not be executed successfully, because there are too many defining unit clauses with which (2b) can unify, if only the argument `Form` is instantiated. Therefore backtracking leads to the selection of one of the other two literals with one argument instantiated, (2c) and (2e). However, upon their selection it turns out that (2c) and (2e) can not be executed successfully either, because there are two defining clauses with which the rhs-literals of their defining clauses unify irrespective of the instantiation of their arguments. This leads to the unsuccessful investigation of all possible orderings of the body of clause (2) and eventually to the failure of `inverse/3`, because their are no reorderings of higher-level clauses that can lead to the approval of the selection of (2b), (2c) and (2e) given an admissible degree of nondeterminacy of 1.

After all the possible orderings of the body of clause (2) have been investigated the chart contains a number of failure-entries.

```
OldRhs:  (2b) aux_verb(P0,P1,Num-in,Pers-in,Form-in,Aux),
         (2c) subj(P1,P2,Num,Pers,Subj-in),
         (2e) obj(P4,P5,_,_,Obj-in),

NewRhs:  (2f) adv(P5,P,[Verb]-out,S-in).
         (2d) main_verb(P3,P4,Form-out,[Subj,Obj]-out,Verb-in),

Chart:
(3)   success([2],adv([often|P],P,[Verb],often(Verb)-in),...)
(4)   success([2],main_verb([see|P],P,pres-in,[Subj,Obj],see(Subj,Obj)),...)
(5)   failure(2,aux_verb(P0,P1,Num,Pers,Form-in,Aux),6)
(6)   failure(2,subj(P1,P2,Num,Pers,Subj-in),2)
(7)   failure(2,obj(P4,P5,_,_,Obj-in),2)
(8)   failure(1,yesnoq(yesnoq(P0,P,askif(S)-in,2))
(9)   failure(0,sentence(P0,P,ques(askif(S))-in,2))
(10)  opened(sentence)
(11)  opened(yesnoq)
```

Chart-entry (8), for example, contains the information that if only the third argument of literal (2) is bound its defining clause can not be inverted without introducing any nondeterminacy, i.e. the minimal degree of nondeterminacy it introduces is 2, which can be used to anticipate the failure of every call with identical binding information and an admissible degree of nondeterminacy of 1, i.e. smaller than 2.

Subsequently, `inverse/3` is called again with the admissible degree of nondeterminacy increased by 1 to 2 and the inversion process by means of using the chart-entries returns to the same situation almost immediately. The failure of the inversion of the clauses defining literal (2b) is anticipated, but both (2c) and (2e) can be executed safely now. As the heuristic rule does not give a preference, one of them is chosen nondeterministically. First, consider the case that (2c) is selected next:

```
OldRhs:   (2b) aux_verb(P0,P1,Num-in,Pers-in,Form-in,Aux),
          (2e) obj(P4,P5,_,_,Obj-in),

NewRhs:   (2f) adv(P5,P,[Verb]-out,S-in).
          (2d) main_verb(P3,P4,Form-out,[Subj,Obj]-out,Verb-in),
          (2c) subj(P1,P2,Num-out,Pers-out,Subj-in),

Chart:
(3)    success([2],adv([often|P],P,[Verb],often(Verb)-in),...)
(4)    success([2],main_verb([see|P],P,pres,[Subj,Obj],see(Subj,Obj)-in),...)
(5)    failure(2,aux_verb(P0,P1,Num,Pers,Form-in,Aux),6)
(6)    failure(2,subj(P1,P2,Num,Pers,Subj-in),2)
(7)    failure(2,obj(P4,P5,_,_,Obj-in),2)
(8)    failure(1,yesnoq(yesnoq(P0,P,askif(S)-in,2))
(9)    failure(0,sentence(P0,P,ques(askif(S))-in,2))
(10)   opened(sentence)
(11)   opened(yesnoq)
(12)   success([2],subj(P1,P2,Num,Pers,Subj-in):-np((P0,P,Num,Pers,su-in,Subj-in).)
```

Note that the original open-entries for `sentence` and `yesnoq` ((1) and (2)) have been retracted and replaced by two new open-entries (9 and (10)). Upon execution of (2c) the arguments `Num` and `Pers` are contained in `NewBoundArgs`. Both (2b) and (2e) can be executed successfully now. Because (2b) has more instantiated arguments, than (2e), it is selected to precede (2e), resulting in the following completely reordered body of clause (2):

```
OldRhs:   -

NewRhs:   (2f) adv(P5,P,[Verb]-out,S-in).
          (2d) main_verb(P3,P4,Form-out,[Subj,Obj]-out,Verb-in),
          (2c) subj(P1,P2,Num-out,Pers-out,Subj-in),
          (2b) aux_verb(P0,P1,Num-in,Pers-in,Form-in,Aux-out),
          (2e) obj(P4,P5,_,_,Obj-in),

Chart:
(3)    success([2],adv([often|P],P,[Verb],often(Verb)-in),...)
```

```
(4)    success([2],main_verb([see|P],P,pres,[Subj,Obj],see(Subj,Obj)-in),...)
(5)    failure(2,aux_verb(P0,P1,Num,Pers,Form-in,Aux),6)
(6)    failure(2,subj(P1,P2,Num,Pers,Subj-in),2)
(7)    failure(2,obj(P4,P5,_,_,Obj-in),2)
(8)    failure(1,yesnoq(yesnoq(P0,P,askif(S)-in,2))
(9)    failure(0,sentence(P0,P,ques(askif(S))-in,2))
(12)   success([2],subj(P1,P2,Num,Pers,Subj-in):-np((P0,P,Num,Pers,su-in,Subj-in).)
(13)   success([2],aux_verb(P0,P1,Num-in,Pers-in,Form-in,Aux),...)
(14)   success([2],obj(P4,P5,_,_,Obj-in):-np((P0,P,Num,Pers,ob-in,Obj-in).)
(15)   success([1],yesnoq(P0,P,askif(S)-in):-
                                     adv([often|P],P,[Verb],often(Verb)-in),...)
(16)   success([0],yesnoq(P0,P,askif(S)-in):-yesnoq(P0,P,askif(S)-in)
```

As the reordering of the clauses defining literal yesnoq(P0,P,askif(S)) is completed now success-entry (14) is added to the chart and the corresponding opened-entry (10) is removed from the chart. If rhs-literal (2e) is selected instead of (2c), the alternative ordering of the body of clause (2) looks as follows:

```
OldRhs:  -

NewRhs:  (2f) adv(P5,P,[Verb]-out,S-in).
         (2d) main_verb(P3,P4,Form-out,[Subj,Obj]-out,Verb-in),
         (2e) obj(P4,P5,_,_,Obj-in),
         (2c) subj(P1,P2,Num-out,Pers-out,Subj-in),
         (2b) aux_verb(P0,P1,Num-in,Pers-in,Form-in,Aux-out),

Chart:
(3)    success([2],adv([often|P],P,[Verb],often(Verb)-in),...)
(4)    success([2],main_verb([see|P],P,pres,[Subj,Obj],see(Subj,Obj)-in,...)
(5)    failure(2,aux_verb(P0,P1,Num,Pers,Form-in,Aux),6)
(6)    failure(2,subj(P1,P2,Num,Pers,Subj-in),2)
(7)    failure(2,obj(P4,P5,_,_,Obj-in),2)
(8)    failure(1,yesnoq(yesnoq(P0,P,askif(S)-in,2))
(9)    failure(0,sentence(P0,P,ques(askif(S))-in,2))
(12)   success([2],obj(P4,P5,_,_,Obj-in):-np((P0,P,Num,Pers,ob-in,Obj-in).)
(13)   success([2],subj(P1,P2,Num,Pers,Subj-in):-np((P0,P,Num,Pers,su-in,Subj-in).)
(14)   success([2],aux_verb(P0,P1,Num-in,Pers-in,Form-in,Aux),...)
(15')  success([1],yesnoq(P0,P,askif(S)-in):-
                                     adv([often|P],P,[Verb],often(Verb)-in),...)
(16)   success([0],yesnoq(P0,P,askif(S)-in):-yesnoq(P0,P,askif(S)-in)
```

The inversion of the grammar fragment results in two new orderings of the body of clause (2) stored in the chart-entries (15) and (15'). Though the EAA finds these orderings as well, it is not able to distinguish them from other less efficient orderings, i.e. which are not optimized as a result of the heuristic rule discussed in 3.2.3.

The overall degree of nondeterminacy introduced by the grammar fragment is 2. The reusage of the intermediate results of the attempt to invert the grammar according to an admissible degree of nondeterminacy of 1 does not lead to a better result in this case (see 3.3.3).

4 TREATMENT OF RECURSIVE CLAUSES

The notion of an *argument series* and its usage to control recursive inference on structural information is discussed. This part of the treatment of recursive clauses in the DIA is similar to the treatment of recursive clauses proposed by Strzalkowski in [16]. However, the actual inversion of recursive clauses in the DIA differs significantly.

In the context of the EAA, the inversion of recursive procedures is supposed to be accomplished by means of, so-called, *interclausal reordering*: a reordering of literals across clause-boundaries. However, Strzalkowski's use of interclausal reordering, as informally described in [16], has as an unacceptable consequence that it changes the semantics of the input-grammar. In the DIA, this is avoided through making use of a reformulation scheme for the inversion of recursive procedures. It is shown, that this reformulation scheme allows the DIA to deal successfully with a particular class of recursive clauses which is known to be hard to deal with in a general manner in a static literal reordering approach. Finally, the presented treatment of recursive clauses is contrasted in more detail with the proposed treatment of recursive clauses in the EAA.

The focus of attention in this section is on procedures with direct recursive clauses with one recursive rhs-literal and a nonrecursive base case. This is done for expository reasons and because the current implementation of the DIA deals only with this subclass of recursive clauses. However, there is nothing that prohibits the generalization of the described treatment for more complex instances of recursion on structural information.

4.1 Argument Series

Just as in the EAA, the treatment of recursive clauses in the DIA is based on the notion of an *argument series* ([16]). Intuitively understood, an argument series is a chain of (partial) instantiations of arguments in a recursive clause between a head-literal argument of the recursive clause and the corresponding argument of a recursive rhs-literal of this clause, i.e. the argument in the same argument position.

Argument series are used to deal with recursion on structural information. They can be used to identify a particular class of recursive procedures for which termination is ensured. Furthermore, they can be used to invert recursive procedures which are not guaranteed to terminate otherwise.

4.1.1 Direct and Indirect Argument Series

Refraining from a detailed description of an argument series for a moment, argument series are described informally. In the DIA two sorts of argument series are used to deal with recursion on structural information: namely, those cases where in a recursive clause the relationship between the arguments of the literals is such that

1. the instantiation of an argument in the head-literal always results in the instantiation of the corresponding argument in the recursive rhs-literal, such that the value of the rhs-literal argument is structurally *less* complex than the value of the head-literal argument. This is the case, when there exists a *peeling argument series* between these arguments.

The relation between the two arguments in the following schematic recursive clause is a stereotypical example of a peeling argument series.

```
q(f(X)):-
   ..,
   q(X),
   ...
```

The instantiation of the head-literal argument always results in a structurally less complex instantiation of the argument of the recursive rhs-literal. Intuitively understood, prior to every recursive call a part of the value of the head-literal argument is “peeled off”, until the base case is reached.

2. The instantiation of an argument in the head-literal always results in the (partial) instantiation of the corresponding argument in the recursive rhs-literal, such that the value of the rhs-literal argument is structurally *more* complex than the value of the head-literal argument. This is the case, when there exists a, so-called, *building argument series* between these arguments.

   The relation between the two arguments in the following schematic recursive clause is a stereotypical example of a building argument series.

```
q(X)):-
    ..,
    q(f(X)),
    ...
```

The instantiation of the head-literal argument always results in a structurally more complex instantiation of the argument of the recursive rhs-literal. Prior to every recursive call new structural information is added. One could say that the recursive clause “builds up” to the base case.

The argument series in the schematic clauses above are *direct* argument series in the sense that the instantiation of the head-literal argument directly results in the (partial) instantiation of the corresponding argument in the recursive rhs-literal. The instantiation of an argument in the head-literal of a recursive clause can also indirectly result in the (partial) instantiation of the corresponding argument in the recursive rhs-literal in which case the argument series is called *indirect* (see also §2.2).

In the following schematic recursive clause there exists an indirect building argument series: the instantiation of the head-literal argument leads to the instantiation of the corresponding argument in the recursive rhs-literal *via* a rhs-literal preceding the recursive rhs-literal.

```
q(X):-
   ..,
   p(X,Y),
   ..,
   q(Y),
   ...

p(X,f(X)).
p(X,g(X)).
```

In order to *participate* in an argument series a rhs-literal preceding the recursive rhs-literal in the recursive clause must exhibit a specific relation between two of its arguments. In the case of the example: the indirect building argument series in the recursive clause depends on the fact that *all* clauses defining the participating rhs-literal when called with the first argument instantiated result in the instantiation of the second argument, such that the value of the first argument is contained in the value of the second argument. As the relation between these arguments can be indirect too, the relation between these two arguments can be considered a sort of argument series as well. As the existence of such a argument series is a prerequisite for there to be an indirect building argument series in the recursive clause, it is called an *auxiliary argument series*.

In case there exists an (direct or indirect) argument series between a head-literal argument and the corresponding argument in the recursive rhs-literal, this argument series need not necessarily be building or peeling. If, for example, the recursive clause has a context argument the value of this argument in the head-literal is always identical to the value of the corresponding argument in the recursive rhs-literal.

4.1.2 Argument Series to Control Recursive Inference

In the overall inversion process, argument series are computed when `inverse_clause/5` has been called for all clauses defining a literal and the chart-entries indicate that one of them is recursive. Before the newly created procedure is stored, it is checked whether the instantiation of the arguments of the literal which instigated the inversion of the clauses ensure the termination of the recursive clause(s) by computing the argument series.

The existence of a (direct or indirect) argument series which is peeling or building, can be used to control the recursion:

1. In case there exists a peeling argument series in a recursive clause, this argument series can be used to determine whether a call to the recursive clause terminates. If a peeling argument series is bound, i.e. the head-literal argument of the argument series is instantiated, at the time the recursive clause is called, the termination of the recursive clause is guaranteed, because it is possible to peel off structural information from the head-literal argument only finitely many times.

2. In case there exists a building argument series in a recursive clause, it is possible to ensure the termination of the recursive clause through turning the building argument series into a peeling argument series by means of *argument series reversal* and subsequently reformulating the recursive procedure as to ensure the newly created peeling argument series is bound and the semantics of the logic grammar is not altered.

A related approach to automated inversion of logic grammars by Dymetman et al. ([3, 4]) makes use of a specific sort of peeling argument series, a *guide*, to ensure termination. The existence of a direct peeling argument series is used in [18] to determine whether recursively defined database queries are guaranteed to terminate.

In case of recursion on nonstructural information, for example, an integer-argument which is decremented down to 0, there are no peeling or building argument series which can be used to control the recursive inference. Though in this case the notion of an argument series might be useful as well, the focus of attention in this section is on controlling recursion on structural information, as this kind of recursion includes an important class of cases of recursion commonly found in logic grammars in which use is made of a lexical encoding of grammatical information (see below).

Before argument series reversal and the reformulation of recursive procedures to support argument series reversal are discussed in more detail, the computation of argument series in the DIA is discussed.

4.1.3 Argument Series Computation

This more detailed description of the notion of an argument series, makes use of the predicates from the DIA-implementation to compute argument series. Again the code has been considerably simplified for expository reasons. The actual code can be found in appendix D. Before discussing the code, the computation of an argument series is explained informally.

Reconsider the example discussed in §4.1.1 and suppose that the literals `p` and `q` are the only rhs-literals in the body of the recursive clause:

```
(1a) q(X):-
(1b)      p(X,Y),
(1c)      q(Y).

(2)  p(X,f(X)).
(3)  p(X,g(X)).
```

It is checked whether (1) is guaranteed to terminate by investigating the relationship between the instantiation of the only head-literal argument of clause (1), i.e. `X` in (1a) and the (partial) instantiation of the corresponding argument in the recursive rhs-literal of (1), i.e. argument `Y` in (1c). This is done by trying to prove an argument series between these two arguments and subsequently checking the relation between the arguments in the argument series.

First, the argument in the head-literal corresponding to the argument position for which an argument series is tried to be proven, is retrieved from the recursive clause. Then an argument in a rhs-literal is sought for which is identical to the retrieved head-literal argument, or a subterm of the head-literal argument, or of which the head-literal argument is a subterm, in which case the argument is said to stand in the *contains*-relation with the head-literal argument. The only argument in the body of (1) which stands in the contains-relation with the argument `X` in (1a) is the argument `X` in (1b), as it is identical to `X` in (1a).

Literal (1b) is not the recursive rhs-literal of (1) and must therefore be a participating rhs-literal. This means that if argument `Y` in the recursive rhs-literal (1c) gets (partially) instantiated as a result of the instantiation of argument `X` in (1a), it gets (partially) instantiated indirectly, namely upon the execution of literal (1b). Therefore there must exist an auxiliary argument series between `X` in (1b) and `Y` in (1b), i.e. the only argument of (1b) besides `X`, in order for there to be an argument series in (1).

Because there exists a direct auxiliary argument series between these two arguments for all clauses defining literal `p`, `Y` is considered the next argument in the argument series in (1). Now an argument in a subsequent rhs-literal is sought for which stands in the contains-relation with argument `Y` in (1b). Argument `Y` in literal (1c) stands in the contains-relation with `Y` in (1b) and as this argument is the argument corresponding to `X` in (1a) an argument series is found. Subsequently, the argument series is identified as building based on the relation between the arguments `X` and `Y` in literal (1b): because the auxiliary argument series in the clauses defining (1b) guarantee argument `X` is a subterm of argument `Y`. As a result of that the instantiation of argument `X` of literal (1a) always leads to a structurally more complex instantiation of argument `Y` of literal (1c).

In order to uniquely identify an argument in a clause during argument series computation, arguments get assigned two indices: `L`, to indicate the literal in which the argument is found (the head-literal corresponds to index 0, and so on) and `A`, to indicate the position of the argument in this literal (the first argument of a literal gets the index 1). So, for example, an argument `X` which is the second argument of the first rhs-literal of a clause is represented as: `X-1-2`.

The predicate `argument_series/2` is called with a recursive clause and a specification of the argument position (`N`) for which an argument series has to be computed.

```
argument_series([[Pred|Args]|Body],N):-
   nth(N,Args,ArgY),
   x_argument([[Pred|Args]|Body],ArgY-0-N,N).
```

The value of the argument at position `N` in the head-literal of the recursive clause is retrieved (`ArgY`) and `x_argument/3` is called to find the next argument (`ArgX`) in the argument series. The position of the head-literal argument is passed along as a context argument to determine in case an argument series is indirect whether the position of the argument in the recursive rhs-literal corresponds to it.

```
x_argument([Head|Body],ArgY-L,N):-
   nth(L1,Body,[BodyPred|BodyArgs]),
   L1 > L,
   nth(A1,BodyArgs,ArgX),
   contains(ArgY,ArgX),
   ( recursive_clause([Head|[[BodyPred|BodyArgs]]]),
     A1 = N, !
   ; y_argument([Head|Body],ArgX-L1-A1,ArgY2-L1-A2),
     x_argument([Head|Body],ArgY2-L1-A2,N)
   ).
```

`x_argument/3` starts looking for an argument in a literal in the body of the clause with which the first argument in the argument series stands in the contains-relation:

```
contains(ArgY,ArgX):-
   ( Argx == Argy
   ; contains_var(Argx,Argy)
   ; contains_var(Argy,Argx)
   ).
```

Depending on the specific relation between the individual arguments in the argument series, the argument series as a whole is of a particular sort (see below). Having found such an argument, it is checked whether the literal in which this argument is found is the recursive rhs-literal (by means of a call to `recursive_clause/1`) and whether the position of the argument in this literal is the same as the position of the first (head-literal) argument, in which case `A1 = N`. If this is the case, a direct argument series is found and the argument series computation for this argument position in the recursive clause is finished. If not, the rhs-literal from which the second argument in the argument series is chosen is a participating literal and it is necessary to find another argument in the same literal with which it stands in the `contains`-relation and prove an auxiliary argument series between them for each of the clauses defining the literal. This is accomplished by the call to `y_argument/3`.

```
y_argument([Head|Body],_ArgX-L1-A1,ArgY-L1-A2):-
   nth(L1,Body,[_BodyPredX|BodyArgs]),
   nth(A2,BodyArgs,ArgYStatus),
   A1 =\= A2,
   auxiliary_argument_series([Head|Body],L1,A1,A2).
```

Subsequently, `x_argument/3` is called again to try and connect the last argument in the argument series (`ArgY2`) to the recursive rhs-literal the same way as was tried to connect the head-literal argument `ArgY` to the recursive rhs-literal.

Briefly returning to the example at the beginning of this section: for the argument position `N = 1` the computation of the indirect argument series proceeds as follows. After retrieving the `N`th argument of the head-literal (`ArgY = X-0-1`), `x_argument/3` is called with the recursive clause, the head-literal argument and its position in the head-literal to seek for an argument which stands in the `contains`-relation with the head-literal. Argument `X-1-1`, which is identical to the head-literal argument, is found and it is checked whether literal `p` equals the recursive rhs-literal of the clause. This is not the case, therefore it is tried to prove auxiliary argument series between `X-1-1` and `Y-1-2` by means of a call to `y_argument/3`. This succeeds as the first and the second argument in literal `p` stand in a `contains`-relation as a result of auxiliary argument series in the unit clauses defining literal `p`. `Y-1-2` is therefore considered the next argument in the argument series. Subsequently, a recursive call to `x_argument/3` leads to the connection of `Y-1-2` with the argument of the recursive rhs-literal `Y-2-1` and an indirect argument series for the argument-position `N` is found.

The above predicates compute argument series correctly, but they do not keep track of the kind of relation between the individual arguments in an argument series. In order to determine whether an argument series as a whole is peeling or building, this infomation is indispensible. In case of a direct argument series, the head-literal argument must be contained in (in case of a building argument series), or contained by (in case of a peeling argument series) the value of the corresponding argument in the recursive rhs-literal, otherwise the argument series is not of interest. However, in case the argument series is indirect, it is also possible that the set of auxiliary series corresponding to a particular participating rhs-literal make an argument series as a whole building or peeling.

In case of the above example, the fact that for all clauses defining literal `p` the first argument is contained in the second argument of the head-literal, makes the argument series as a whole building. If in one of the defining unit clauses the first and the second argument would have been identical to each other, the argument series in the recursive clause would not have been considered building: the instantiation of the head-literal argument does not necessarily lead to an instantiation of the argument in the recursive rhs-literal which is structurally more complex.

The notion of an argument series as used in the DIA is not completely identical to the one proposed in the context of the EAA: the definition of an argument series is generalized with respect to the defintion in [16] in that the binding status of the individual arguments in an argument series is completely ignored. It is possible to abstract over the binding status of arguments in an argument series, because in case of a building or peeling argument series the fact that the `contains`-relation holds between all arguments in the argument series guarantees that they are instantiated appropriately. Adopting Strzalkowski's definition of an argument series would complicate the computation of an argument series and furthermore necessitate an extension of the set of rules used in the DIA to compute the status of an argument as discussed in §3.2.2.

4.2 Argument Series Reversal and Reformulation

In case a recursive procedure is called such that its recursive clause(s) do(es) not have an instantiated peeling argument series, use is made of the reversal of a building argument series which in combination with the reformulation of the procedure allows to control the recursive inference. This is illustrated by means of a concrete example. The Prolog predicates used for the reversal of argument series and the reformulation of recursive procedures can be found in appendix E and appendix F, respectively.

```
(1a) vp(P0,P,Args,VSem):-
(1b)     vp(P0,P1,[CSem|Args],VSem),
(1c)     np(P1,P,CSem).
(2a) vp(P0,P,Args,VSem):-
(2b)     v(P0,P,Args,VSem).

(3)  v([chased|P],P,[Obj,Subj],chased(Subj,Obj)).
     .........................
(4)  np([john|P],P,john).
(5)  np([fido|P],P,fido).
     .........................
```

The above grammar extract[12] is chosen as an example, because clauses like (1), which are also called *deadlock-prone clauses*, are discussed in [12] as an example of clauses that lead to the nontermination of top-down generation algorithms (independent of the ordering of their body) for which no general solution can be found due to the fact that the recursion controlling information is encoded in the unit clauses constituting the lexicon. It is shown that by means of argument series reversal in combination with the reformulation of the recursive procedure these clauses can be dealed with successfully in the DIA.

Upon the reordering of its rhs-literals this recursive procedure is checked for argument series and the fact that it has no instantiated peeling argument series leads to an investigation of all argument postions to determine whether there exists a building argument series. In this case there exists a building argument series for argument position three: if `Args` is instantiated, the third argument of the next recursive call to clause (1) has a structurally more complex value. The third argument is build up to its base case value in unit clause (3).

This building argument series is changed into a peeling argument series by means of argument series reversal, in this case this leads to swapping the arguments `Args` and `[CSem|Args]` in the literals (1a) and (1b), respectively, resulting in the following adapted recursive `vp`-procedure[13].

[12] This grammar extract is also discussed in [16].

[13] Notice that performing argument series reversal to obtain a new peeling series, might involve the swapping of more than one pair of arguments: every pair of successive arguments in an argument series, where the first is contained in the second have to be swapped. In case of an indirect building argument series, it is possible that the arguments in the same (participating) rhs-literal have to be swapped (see also 4.3).

```
(1a') vp_rec(P0,P,[CSem|Args],VSem):-
(1b')    vp_rec(P0,P1,Args,VSem),
(1c')    np(P1,P,CSem).
(2a') vp_rec(P0,P,Args,VSem):-
(2b')    v(P0,P,Args,VSem).
```

In order to use the adapted recursive vp-procedure to control the recursive inference, it must be reformulated, to ensure that the newly created peeling argument series is bound at the time of execution and to ensure that the semantics of the grammar remains unchanged. The reformulation consists of three steps:

1. Adding a new clause which substitutes the recursive procedure. This clause retrieves the base case value of the argument corresponding to the peeling argument series, i.e. the recursion controlling information, by means of "moving up" the base case call of the recursive procedure (see below). Subsequently, the further adapted recursive procedure (see below) is called with the recursion controlling information as the value of the argument corresponding to the newly created peeling series. The results of the moved-up base case call and the original input-value of the argument corresponding to the reversed argument series are wrapped together and passed along the recursive procedure to ensure that the recursion is constrained appropriately and the results of the base case call end up where they belong.

2. Moving up the base case call of the recursive procedure is done by means of isolating the nonrecursive clauses of the recursive procedure and giving them a new predicate name in order to enable the retrieval of the recursion controlling information directly, without using the recursive clause.

3. After a peeling series is created, the recursive procedure is further adapted by adding an extra (context) argument to pass along the original input-value of the argument corresponding to the peeling argument series to make sure the recursion is constrained appropriately, and the output-arguments of the new moved up base case call to identify them with the arguments of the former base case call.

In case of the above example the new intermediate clause looks as follows:

```
vp(P0,P,Args,VSem):-
   vp_base(NewP1,NewP2,NewArgs,VSem),
                        % moved up base case call
   vp_rec(P0,P,NewArgs,VSem,bundle(NewP1,NewP2,Args,VSem)).
                        % feeding the output-arguments
                        % of the base case call into the call to
                        % the new recursive procedure with
                        % the reversed argument series.
```

The new clause retrieves the base case value of the argument corresponding to the reversed building argument series, by means of the call to the predicate vp_base:

```
vp_base(P0,P,Args,VSem):-
   v(P0,P,Args,VSem).
```

The vp_base-clause is the isolated base case of the former recursive vp-procedure (clause (2) above)[14]. Instead of using the third argument of vp_base as an input-argument as in clause (2), it is used as the output-argument providing the recursion controlling information to be fed into the call to the adapted recursive procedure with the reversed building argument series (vp_rec) as the value of the argument corresponding to the newly created peeling series (the argument in position three).

In order to constrain the calls to vp_base and vp_rec the original input-argument VSem is used. An original input-argument can be used to constraint the base case call, when its value is left unchanged by the recursive clause. If the recursive clause would, for example, add structural information to the value of an argument, then the value of that argument can not be used to constrain the base case call, because no structural information has been added when the base case call takes place. An argument series can be used to determine whether an argument is changed by a recursive clause or contextual: an argument is contextual if all arguments in an argument series are identical[15].

The other arguments of vp_base (NewP1, NewP2 and VSem) are wrapped together with the original input-argument corresponding to the reversed argument series (Args) and passed along the vp_rec-procedure as its newly added fifth argument (Bundle):

```
(1a'') vp_rec(P0,P,[CSem|Args],VSem,Bundle):-
(1b''     vp_rec(P0,P1,Args,VSem,Bundle),
(1c'')    np(P1,P,CSem). % the recursive clause with the reversed
                         % building argument series
(2a'') vp_rec(P0,P,Args,VSem,bundle(P0,P,Args,VSem)).
                         % check peeled off base case against
                         % the original input-arguments
```

By substituting the predicates resulting from reformulation for the clauses (1) and (2), the termination of a call to vp with only the argument VSem instantiated is now ensured to terminate.

It is important to notice that in case a recursive procedure contains more than one recursive clause the argument series reversal and reformulation of the recursive procedure can be more complicated. In the DIA-implementation the assumption is that either all recursive clauses have an instantiated peeling argument series or there exists a reversible building argument series for one and the same argument-position in all the recursive clauses in the procedure.

[14] For each base case of the recursive procedure under reformulation there is a new vp_base-clause added. As clause (2) is the only base case of the vp-procedure, there is only one vp_base-clause in the new reformulated grammar extract.

[15] Note that the argument VSem can only be used to constrain the base case call if the recursive clause is a *chain rule*, as in [12]: a rule in which the semantics of some right-hand side element is identical to the semantics of the left-hand side. On the one hand, the use of arguments to constrain the base case call is a generalization of the notion of a chain rule, because there is no restriction to the argument carrying the semantics. On the other hand, a chain rule as used in [12] can be nonrecursive.

Consider the following abstract example of a recursive procedure containing two recursive clauses which violates the above assumption:

```
(1) q(f(X)):-
        ...,
        q(X).
(2) q(X):-
        ...,
        q(f(X)).
(3) q(X):-
        p(X).
```

This recursive procedure is problematic, because in (1) there exists a peeling series, while in (2) there exists a building series. The series reversal and reformulation discussed above does not suffice to guarantee the termination of a call to this recursive procedure, because only reversing the argument series in (2) changes the semantics of the procedure. However, up to now no natural language examples were encountered which seem to justify such combination of recursive clauses.

4.3 Comparison with the Treatment of Recursive Clauses in the EAA

There are a four important differences between the treatment of recursive clauses in the DIA and the treatment of recursive clauses as proposed in the context of the EAA.

1. According to the proposals in [16], instead of the reformulation of a recursive procedure a reversed building argument series should trigger, so-called, *interclausal reordering*: a technique akin to partial execution ([9]) which accomplishes a reordering of literals across clause boundaries. The suggested use of interclausal reordering leads to a change of the meaning of the grammar. Strzalkowski does not notice this unacceptable consequence of his treatment of recursive clauses. Furthermore, it is completely unclear how interclausal reordering, which is only described informally, should be incorporated in the *intraclausal* reordering strategy outlined in section 2.

2. In the DIA the binding status of the arguments in the grammar does not influence the identification of the relevant argument series. This means that the set of rules discussed in §3.2.2 supports all the binding information that needs to be available. The definition of an argument series in [16] makes use of the binding status of arguments, which necessitates an extension of the discussed rules to determine when an argument in a literal is partially instantiated upon the execution of this literal.

3. In the EAA argument series reversal is supposed to take place as part of the precalculation of MSEAs, but only in case it is impossible to find a blocker for a recursive clause, because it is impossible to reverse every building series in the grammar in advance. However, finding a blocker for a recursive clause does not mean that this blocker actually becomes instantiated and if a blocker is not instantiated, it does not block the recursion. So, for example, for a recursive clause with the head-literal `lit(A,B,C)` the argument `A` can not be determined to block the recursion without any binding information. If `A` is bound the recursion terminates, but if `A` is not bound, termination can not be guaranteed and it is necessary to find another way to block the recursion. This means that all recursive clauses

which have a blocker that never becomes instantiated are dismissed as irreversible even though it might be the case that there is a building argument series that can be reversed successfully. Due to its goal-directedness, the DIA successfully inverts these recursive clauses, because it is known which arguments are bound at the moment it is decided whether a building argument series should be reversed.

4. In contrast to the proposals in [16] indirect argument series reversal takes place within the recursive clause and not in the clauses defining the participating rhs-literal. This has as an advantage that the defining clauses do not have to be copied.

Although the question of whether a recursive call terminates is in general undecidable and does not dispose of further restrictions that need to be placed on the grammar, it is shown that the presented treatment of recursive clauses is able to deal with an important class of cases of recursion on structural information, typically used in grammars which make use of lexical encoding of grammatical information such as in categorial grammar, in head-driven phrase-structure grammar and in lexical functional grammar.

5 CONCLUDING REMARKS

A new approach to the automated inversion of logic grammars, the DIA, was described and compared to the related EAA. The DIA improves upon the EAA in being goal-directed, which makes it a feasible approach to the automated inversion of logic grammars from a computational point of view. In addition, the class of grammars that can be inverted is increased and the quality of the new executable grammar is better for a specific class of grammars.

By means of argument series reversal in combination with the reformulation of recursive procedures, the DIA deals successfully with the inversion of logic grammars which contain deadlock-prone clauses, clauses which were suspected impossible to deal with in a general manner such that a top-down control strategy is guaranteed to terminate.

In their comparison of the EAA with the Semantic Head-Driven Generation Approach (SHDGA; [12]), Martinovic and Strzalkowski ([7]) rely quite heavily upon the existence of a general treatment of deadlock-prone clauses. But the treatment of deadlock-prone clauses in the context of the EAA suggested in [16] can hardly be considered satisfactorily, if only for the fact that the suggested treatment implies a change in the semantics between the input-grammar and the new executable grammar produced. However, having shown that there does exist a general treatment of deadlock-prone clauses, with respect to the DIA the advantages over the SHDGA with respect to efficiency and generality alluded to in [7] seem well-founded.

Future research will focus on the generalization of the DIA. During the inversion of a logic grammar only the effect of fully instantiated arguments on the successful execution of literals is used. The DIA can be significantly improved if the rules to compute the status of an argument could be extended as to deal with arguments which are *properly* instantiated [10]: for example, a list is properly instantiated if it is the empty list `[]`, or if it is `[Head|Tail]`, where `Head` is any term and its `Tail` is a properly instantiated list. As it stands, it is not possible to use the effect of properly instantiated arguments on the execution of literals, due to the lack of a general definition of the notion of a properly instantiated argument.

Currently, the DIA is able to deal only with recursion on structural information, which exhibits either an instantiated peeling argument series or a reversible building argument series. It would be interesting to investigate whether the treatment of recursive clauses could be extended such as to deal with other sorts of linguistically relevant recursion.

APPENDIX

Though the predicates discussed in the forgoing part of the report are simplified, their predicate names correspond to the names of the predicates used in the various parts of the appendix.

The division of the code over the various parts of the appendix reflects the module-structure of the implementation:

- Part A contains the predicates that take care of the conversion of an input-grammar into the representation used in the inversion process and the conversion of the internal representation of the new executable grammar to the output-grammar.
- Part B contains the predicates of the main inversion procedure. The top-level predicate of the inversion process `inversion/4` is given and it is indicated how it should be called.
- Part C contains the DIA-implementation of the rules to compute the binding status of the arguments in the input-grammar.
- Part D contains the predicates for (auxiliary) argument series computation and interpretation.
- Part E contains the predicates for argument series reversal. Some of the predicates are similar to the predicates used to compute an argument series. The main difference being that in this case an argument series is not only computed, but if possible at the same time reversed. The correspondence between these predicates is indicated by maintaining the same predicate name as the one in the previous appendix with the suffix `_reverse`.
- Part F contains the predicates for the reformulation of a recursive procedure in which argument series reversal has already created a new peeling argument series.

The DIA-implementation makes limited use of the Quintus library-predicates:

```
:- use_module(library(occurs),[contains_var/2]).
:- use_module(library(sets),[select/3,intersection/2]).
:- use_module(library(lists),[nth1/3,same_length/2,keys_and_values/3,rev/2]).
:- use_module(library(samsort),[sort/2]).
:- use_module(library(basics),[member/2]).
:- use_module(library(more_maps),[include/3]).
```

A I/O

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Internally, the input-grammar is represented by a list of clauses.  Various
% parts of the clauses have been extended:
% 1.  Clause_index:  each clause is extended with an index to uniquely identify a
% clause (an integer);
% 2.  Status-index:  each argument of the literals in a clause is extended with an
% index which is later used to mark whether and when an argument is considered
% bound (a variable);
% 3.  Copy-index:  an index (0) added to the predicate-name to facilitate the
% copying of a procedure during the inversion process.  So, for example, if the
% top-level predicate name is 'sentence' in the original input-grammar, it gets
% assigned the new predicate name 'sentence-0'.  Every time a new copy of a
% predicate is made the index is increased by one.  The predicates in the
% input-grammar are considered the original version and therefore get the index
% 0.
% By means of example, the internal representation of the clauses:
% sentence(P0,P,Sem):-
%  np(P0,P1,Ssem),
%  vp(P1,P,[Ssem],Sem).
% np([john|P],P,john).
% looks as follows:
% [
% [[sentence-0,P0-_Status,P-_Status,Sem-_Status],
%     [np-0,P0-_Status,P1-_Status,SSem-_Status],
%     [vp-0,P1-_Status,P-_Status,[Ssem-_Status]-_Status]-1,
% [[np-0,[john-_Status|P-Status],P-_Status,john-_Status]]-2
% ]
% The conversion from the internal representation of the new executable grammar
% to the actual output-grammar is essentially the inverse from the conversion
% discussed above.  The main differences being that the new grammar is internally
% represented not as a list, but by means of success-entries in the chart.

:- dynamic success/2,failure/3,opened/1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% read_input_grammar( + InputGrammar, - InternalRepInputGrammar)
read_input_grammar(InputGrammar,I):-
   seen,
   see(InputGrammar),
   read_grammar(1,I),
   seen.

read_grammar(N0,[FirstClause1-N0|RestI]):-
   read(Clause),
   \+ Clause = end_of_file,
```

```
   !,
   ( Clause = (Head:-Body) ->
     Head =..  HeadList,
     add_status(HeadList,HeadList1),
     seq_to_list(Body,BodyList),
     add_status_inds(BodyList,BodyList1),
     FirstClause = [HeadList1|BodyList1]
   ; Clause =..  HeadList,
     add_status(HeadList,HeadList1),
     FirstClause = [HeadList1]
   ),
   copy_ind(FirstClause,FirstClause1),
   N1 is N0 + 1,
   read_grammar(N1,RestI).
read_grammar(_,[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The new executable grammar is written to the terminal as well as to an
% external file OutputGrammar.
% write_output_grammar( + OutputGrammar, + TopGoal, + DegreeofNondeterminacy)
write_output_grammar(OutputGrammar,TopGoal,Deg):-
   open(OutputGrammar,write,P),
   write_degree(Deg),
   write('Output-grammar:'),nl,
   write_def(P,TopGoal),
   !,
   close(P).

write_degree(Deg):-
   nl,
   write('Maximal degree of nondeterminacy:  '),
   write(Deg),
   nl,nl.

write_def(P,[PredX|Args]):-
   retract((success(_,[[[PredX|OldArgs]|Body]-_|RestProc]-_))),
   same_length(Args,OldArgs),
   write_proc(P,[[[PredX|OldArgs]|Body]-_|RestProc],AllBodies),
% The argument Allbodies is used to determine which success-entries in the chart
% belong to the output-grammar.
   write_bodies(P,AllBodies),
   fail.
write_def(_,_).

write_bodies(_,[]).
write_bodies(P,[F|R]):-
   write_body_def(P,F),
```

```
   write_bodies(P,R).

write_body_def(_,[]).
write_body_def(P,[F|R]):-
   write_def(P,F),
   write_body_def(P,R).

write_proc(_,[],[]).
write_proc(P,[[HeadI|BodyI]-_|Clauses],[BodyI|Bodies]):-
   add_status_inds([Head|Body],[HeadI|BodyI]),
   numbervars([Head|Body],0,_),
   write_lhs(P,Head),
   ( Body = [] ->
     write('.'),
     nl,
     write(P,'.'),
     nl(P)
   ; write(':-'),
     write(P,':-'),
     write_rhs(P,Body),
     nl,
     nl(P)
   ),
   write_proc(P,Clauses,Bodies).

write_rhs(P,[F|R]):-
   nl,nl(P),
   tab(3),write(P,'   '),
   write_lhs(P,F),
   ( R = [] ->
     write('.'),
     write(P,'.')
   ; write(','),
     write(P,',')
   ),
   write_rhs(P,R).
write_rhs(_,[]).

% The predicate names of the predicates resulting from reformulation carry
% besides a copy-index also type-information which indicates what part of the
% original recursive procedure they correspond to.
write_lhs(P,[Pred-Type-Index|L]):-
   !,
write(Pred),write(P,Pred),
   write('_'),write(P,'_'),
   write(Type),write(P,Type),
   write('_'),write(P,'_'),
```

```
   write(Index),write(P,Index),
   write('('),write(P,'('),
   write_list(P,L),
   write(')'),write(P,')'),
   !.
write_lhs(P,[Pred-Index|L]):-
   write(Pred),write(P,Pred),
   write('_'),write(P,'_'),
   write(Index),write(P,Index),
   write('('),write(P,'('),
   write_list(P,L),
   write(')'),write(P,')'),
   !.
write_lhs(P,L):-
   T =..  L,
   write(T),write(P,T),
   !.

write_list(P,[O]):-
   write(O),write(P,O).
write_list(P,[F|R]):-
   write(F),write(P,F),
   write(','),write(P,','),
   write_list(P,R).

% Converts a sequence to a list.
seq_to_list((F,R),[F1|R1]):-
   !,
   F =..  F1,
   seq_to_list(R,R1).
seq_to_list((F),[F1]):-
   !,
   F =..  F1.
seq_to_list(_,[]).

% add_status_inds/2 is used to add or remove status-indices from a list of
% clauses.
add_status_inds([F|R],[F1|R1]):-
   add_status(F,F1),
   add_status_inds(R,R1).
add_status_inds([],[]).

add_status([PredX|Args],[PredX|NewArgs]):-
   keys_and_values(NewArgs,Args,_).

% Arg is a subterm of an argument that is 'in' or 'out' in a list of arguments.
in_out_member(Arg-Status,[A-Status|_Args]):-
```

```
   contains_var(Arg,A),
   ( Status == in
   ; Status == out
   ),
   !.
in_out_member(Arg,[_|Args]):-
   in_out_member(Arg,Args).

member_prefix(X,[X|_],[]).
member_prefix(X,[Z|Y],[Z|P]):-
   member_prefix(X,Y,P).

% Removes or adds copy-indices from or to the predicate names in the
% input-grammar.
copy_index([],[]).
copy_index([F-Ind|R],[NF-Ind|NR]):-
   copy_ind(NF,F),
   copy_ind(R,NR).

copy_ind([[Pred|Args]],[[Pred-0|Args]]).
copy_ind([[Pred|Args]|R],[[Pred-0|Args]|R1]):-
   copy_ind(R,R1).
```

B MAIN INVERSION PROCEDURE

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% inversion/4 is the top-level predicate:  inversion( + Degree, + Inputgrammar, +
% TopGoal, + Outputgrammar).  Where Degree is a positive integer which specifies
% the maximal degree of nondeterminacy that the new executable grammar is allowed
% to have.  Inputgrammar and Outputgrammar are the file-specifications of the
% files where the input-grammar can be found and the output-grammar has to be
% stored, respectively.  (The output-grammar is also written to the terminal.)
% TopGoal specifies the top-level predicate of the input-grammar and its intended
% input-arguments by means of the status-index 'in', e.g.  [sentence-0,_,_,_-in].
% Thus inversion(3,input_grammar,[sentence-0,_,_,_-in],output_grammar).  is true,
% if there exists an inversion of input_grammar, output_grammar, with a maximal
% degree of nondeterminacy of three in case the third argument of the top-level
% predicate sentence-0 is instantiated.  Prior to the call to determinacy_loop/4
% the chart is cleaned up.

inversion(Max,InputGrammar,TopGoal,OutputGrammar):-
   read_input_grammar(InputGrammar,I),
   retractall(failure(_,_,_)),
   retractall(success(_,_)),
   determinacy_loop(Max,1,TopGoal,I,Deg),
   write_output_grammar(OutputGrammar,TopGoal,Deg).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For every degree of determinacy (up to and including Max) inverse/6 is called
% with the top-level predicate of the input-grammar by means of an iterative
% deepening technique.  The internal representation of the input-grammar is passed
% along by means of an accumulator pair.  Before each consecutive call to
% inverse/6 irrelevant chart-entries are removed.  The argument NewPredName is
% used to accomplish the renaming of the call to a predicate that is copied and
% renamed during the inversion process.
determinacy_loop(Max,D,TopGoal,I0,Deg):-
   retractall(opened(_)),
   ( inverse(D,TopGoal,_,0,I0,_), Deg = D
   ; (D < Max ->
     DNew is D+1,
     determinacy_loop(Max,DNew,TopGoal,I0,Deg))
   ).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The failure-entries in the chart are used for fail anticipation.  If the
% predicate no_fail_anticipation/4 fails a fail is anticipated and the inversion
% of the clauses defining literal [Pred-X|Args] fails before they have been
% retrieved from the internal representation of the input-grammar.
inverse(D,[Pred-X|Args],PredZ,Ind,I0,I):-
   failure(_,[[[Pred-_|OldArgs]|_]-_]-_,N),
   same_length(Args,OldArgs),
   !,
   no_fail_anticipation(OldArgs,Args,N,D),
   inverse_s(D,[Pred-X|Args],PredZ,Ind,I0,I).
inverse(D,[PredX|Args],PredZ,Ind,I0,I):-
   inverse_s(D,[PredX|Args],PredZ,Ind,I0,I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The chart is consulted whether it is useful to invert the current procedure.
% It is checked whether the clauses defining [Pred-X|Args] were previously
% successfully inverted and if so can this inversion be reused (determined by
% reorder/3).  If this is not the case inv/6 is called to retrieve the defining
% clauses and mark them with the new binding information one by one.  The
% chart-entries carry an index indicating whether the defining clauses are all
% unit clauses (fac) or not (def).  The second clause determines whether it is
% necessary to check the determinacy of the defining unit clauses, as in this
% case no reordering is to be performed.  inverse_s/6 also updates the chart.
inverse_s(_,[Pred-_X|Args],Pred-Z,Ind,I,I):-
   success(Indices0,[[[Pred-Y|OldArgs]|Body]-Index|Rest]-def),
   same_length(OldArgs,Args),
   reorder(OldArgs,Args,Result),
   Result == no-reorder,
   !,
   ( \+ member(Ind,Indices0) ->
```

```
      retract((success(Indices0,[[[Pred-Y|OldArgs]|Body]-Index|Rest]-def))),
      append([Ind],Indices0,Indices),
      asserta((success(Indices,[[[Pred-Y|OldArgs]|Body]-Index|Rest]-def)))
    ; true
    ),
    Pred-Y = Pred-Z.
inverse_s(_,[PredX|Args],PredX,Ind,I,I):-
    success(Indices0,[[[PredX|OldArgs]|Body]-Index|Rest]-fac),
    same_length(OldArgs,Args),
    ins_properly_contained_in(OldArgs,Args),
    !,
    ( \+ member(Ind,Indices0) ->
      retract((success(Indices0,[[[PredX|OldArgs]|Body]-Index|Rest]-fac))),
      append([Ind],Indices0,Indices),
      asserta((success(Indices,[[[PredX|OldArgs]|Body]-Index|Rest]-fac)))
    ; true
    ).
inverse_s(D,[PredX|Args],PredZ,Ind,I0,I):-
    inv(D,[PredX|Args],PredZ,Ind,I0,I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The chart is used to check whether the clauses defining [PredX|Args] are opened
% for inversion already by means of checking whether the chart contains an
% opened-entry for PredX. If so the defining clauses are directly or indirectly
% recursive and have to be treated alternatively (to avoid a nonterminating
% inversion process).
inv(_,[PredX|_],PredX,_,I0,I0):-
    opened(PredX),
    !.
% Retrieves the defining clauses of a literal from the internal representation of
% the input-grammar (by procedure/3 and evidence/3).
inv(D,[PredX|Args],PredZ,Ind,I0,I):-
    procedure([PredX|Args],Proc0,I0),
    !,
    inversion_procedure(D,[PredX|Args],Ind,Proc0,PredZ,I0,I).
inv(D,[PredX|Args],PredX,Ind,I,I):-
    evidence([PredX|Args],DefCl,I),
    !,storage_evidence(D,[PredX|Args],Ind,DefCl).

% The determinacy of the call given the current binding information is checked
% (degree_of_nondeterminacy/3) and if OK nonunit clauses are inverted one by one
% by inv_procedure/8 after an opened-entry is added to the chart to indicate that
% this procedure is currently in the process of being inverted.  In case
% inv_procedure/8 fails for a procedure this is stored in the chart by means of
% asserting failure-entry in the database.  degree_proc/4 determine the minimal
% degree of nondeterminacy introduced by the nonunit clauses in the procedure
% defining a literal which is to be used for fail-anticipation.
```

```
inversion_procedure(D,[PredX|Args],Ind,Proc0,PredZ,I0,I):-
   asserta((opened(PredX))),
   degree_of_nondeterminacy(D,[PredX|Args],Proc0),
   initialize([PredX|Args],Proc0,Proc,I0,I1),
   inv_procedure(Proc,D,PredZ,[],_,Ind,I1,I),
   !.
inversion_procedure(D,[PredX|Args],Ind,Proc0,_,_,_):-
   retract((opened(PredX))),
   ( retract((failure(Ind,[[[PredX|Args]|_]-_]-def,Degree)))
   ; true
   ),
   degree_proc(Proc0,PredX,Degree,D),
   !,
   assert((failure(Ind,[[[PredX|Args]|_]-_]-def,Degree))),
   fail.

% The determinacy of the call given the current binding information is checked
% (degree_of_nondeterminacy/3) and if OK unit clauses are stored if not stored
% before (checked by assertion_check/3).  If too much nondeterminacy is introduced
% a failure-entry is added to the chart.  degree_evid/4 determine the actual
% degree of nondeterminacy introduced by the unit clauses defining [PredX|Args]
% which is to be used for fail-anticipation.
storage_evidence(D,[PredX|Args],Ind,DefCl):-
   degree_of_nondeterminacy(D,[PredX|Args],DefCl),
   !,
   assertion_check(D,[PredX|Args],DefCl,Ind).
storage_evidence(D,[PredX|Args],Ind,DefCl):-
   degree_evid(D,[PredX|Args],DefCl,Degree),
   !,
   asserta((failure(Ind,[[[PredX|Args]|_]-_]-fac,Degree))),
   fail.

% Checks whether the unit clauses defining a literal have to be asserted to the
% chart as a success-entry.
assertion_check(_,[PredX|Args],_,_):-
   success(_,[[[PredX|OldArgs]|_]-_|_]-fac),
   same_length(OldArgs,Args),
   !.
assertion_check(_,_,DefCl,Ind):-
   asserta((success([Ind],DefCl-fac))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% inv_procedure/8 calls inv_clause/11 for all defining clauses.  When there are
% no more defining clauses left the new procedure is stored by results/4.
inv_procedure([],_,PredZ,Proc,Proc,Ind,I,I):-
   results(PredZ,Proc,Ind,I),
   !.
```

```
inv_procedure([[[PredX|Args]|Body]-Index|RProc],D,PredZ,Proc0,Proc,Ind,I0,I):-
   include(status_in,Args,BArgs),
   inv_clause(Body,D,[PredX|Args],PredZ,[],BArgs,Index,Proc0,Proc1,I0,I1),
   inv_procedure(RProc,D,PredZ,Proc1,Proc,Ind,I1,I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The actual reordering of the body of a defining clause in accordance with the
% binding information.  This reordering process is completed in case OldRhs is
% empty and the new clause is subsequently stored in P the representation of the
% newly created procedure.
inv_clause([],_,Head,_,NewRhs,_,Ind,P0,P,I,I):-
   append(P0,[[Head|NewRhs]-Ind],P),
   !.
inv_clause(OldRhs,D,Head,PredZ,NewRhs,BoundArgs,Ind,P0,P,I0,I):-
   bound_args_oldrhs(OldRhs,BoundArgs),
   select_rhs_literal(OldRhs,[PredSelLit|ArgsSelLit],OldRhsSub),
   inverse(D,[PredSelLit|ArgsSelLit],NPredSelLit,Ind,I0,I1),
   append(NewRhs,[[NPredSelLit|ArgsSelLit]],NewestRhs),
   bound_args_newrhs(OldRhsSub,Head,NewestRhs,NBoundArgs,Ind,I1),
   inv_clause(OldRhsSub,D,Head,PredZ,NewestRhs,NBoundArgs,Ind,P0,P,I1,I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% After a procedure has been completely inverted results/4 is called by
% inv_procedure/8 to store the new procedure as a success-entry in the chart.  In
% case of a recursive procedure it is checked whether given the associated
% binding information the recursion can be guaranteed to terminate
% (termination_ensured/2).  If so the procedure is treated as a nonrecursive
% procedure.  If not it is necessary to rely on the reversal of a building
% argument series and the reformulation of the recursive procedure
% (building_series_reverse/4).  If a procedure has been inverted previously it is
% checked whether the new inversion is more restrictive.  (It is never less
% restrictive because the inversion process is not performed in that case.)  If
% the new inversion is more restrictive it is going to replace the previous
% inversion.
results(Pred-Y,[[[Pred-X|Args]|Body]-Ind|RProc],Index,I):-
% The new binding information is more restrictive.  The procedure is recursive.
   success(Indices0,[[[Pred-Y|ArgsOld]|_]-_|_]-def),
   same_length(Args,ArgsOld),
   ins_properly_contained_in(Args,ArgsOld),
   recursion([[[Pred-X|Args]|Body]-Ind|RProc],NonRCls,RCls),
   \+ termination_ensured(RCls,I),
   !,
   ( building_series_reverse(NonRCls,RCls,NewProcs,I),
     new_name_ref_proc(NewProcs,Pred-Y,CopyProc),
     (  \+ member(Ind,Indices0) ->
       append([Index],Indices0,Indices),
       asserta((success(Indices,CopyProc-def)))
```

```
      ;  asserta((success(Indices0,CopyProc-def)))
      ),
      retract((success(Indices0,[[[Pred-Y|ArgsOld]|_]-_|_]-def))),
      retract((opened(Pred-X)))
   ; asserta((failure(Ind,[[[Pred-X|Args]|_]-_]-def,0))),
      fail
   ).
results(Pred-Y,[[[Pred-X|Args]|Body]-Ind|RProc],Index,_):-
% The new binding information is more restrictive.  The procedure is not
% recursive.
   success(Indices0,[[[Pred-Y|ArgsOld]|_]-_|_]-def),
   same_length(Args,ArgsOld),
   ins_properly_contained_in(Args,ArgsOld),
   new_name_proc([[[Pred-X|Args]|Body]-Ind|RProc],Pred-Y,CopyProc),
   retract((success(Indices0,[[[Pred-Y|ArgsOld]|_]-_|_]-def))),
   ( \+ member(Ind,Indices0) ->
      append([Index],Indices0,Indices),
      asserta((success(Indices,CopyProc-def)))
   ; asserta((success(Indices0,CopyProc-def)))
   ),
   retract((opened(Pred-X))).
results(Pred-Z,[[[Pred-X|Args]|Body]-Ind|RProc],Index,I):-
% The new and the old inversion are not comparable.  The procedure is
% recursive.
   success(_,[[[Pred-_|ArgsOld]|_]-_|_]-def),
   same_length(Args,ArgsOld),
   recursion([[[Pred-X|Args]|Body]-Ind|RProc],NonRCls,RCls),
   \+ termination_ensured(RCls,I),
   !,
   ( building_series_reverse(NonRCls,RCls,NewProcs,I),
      retract((opened(Pred-X))),
      new_pred_name(Pred-0,Pred-Z),
      new_name_ref_proc(NewProcs,Pred-Z,RenamedNewProcs),
      asserta((success([Index],RenamedNewProcs-def)))
   ; asserta((failure(Ind,[[[Pred-X|Args]|_]-_]-def,0))),
      fail
   ).
results(Pred-Z,[[[Pred-X|Args]|Body]-Ind|RProc],Index,_):-
% The new and the old inversion are not comparable.  The procedure is not
% recursive.
   success(_,[[[Pred-_|ArgsOld]|_]-_|_]-def),
   same_length(Args,ArgsOld),
   new_pred_name(Pred-0,Pred-Z),
   new_name_proc([[[Pred-X|Args]|Body]-Ind|RProc],Pred-Z,RenamedProc),
   asserta((success(Index,RenamedProc-def))),
   retract((opened(Pred-X))).
results(PredX,[[[PredX|Args]|Body]-Ind|RProc],Index,I):-
```

```
% There has been no previous inversion of this recursive procedure.
   recursion([[[PredX|Args]|Body]-Ind|RProc],NonRCls,RCls),
   \+ termination_ensured(RCls,I),
   !,
   ( building_series_reverse(NonRCls,RCls,NewProcs,I),
     retract((opened(PredX))),
     asserta((success([Index],NewProcs-def)))
   ; asserta((failure(Ind,[[[PredX|Args]|_]-_]-def,0))),
     fail
   ).
results(PredX,[[[PredX|Args]|Body]-Ind|RProc],Index,_):-
% There has been no previous inversion of this nonrecursive procedure.
   asserta((success([Index],[[[PredX|Args]|Body]-Ind|RProc]-def))),
   retract((opened(PredX))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% recursion/3 is true if Procedure is a directly recursive procedure.  Currently
% the chart is used to avoid a nonterminating inversion process, but as is dealt
% with direct recursive procedures only, the chart is not used to determine
% whether a procedure is recursive.
% recursion( + Procedure, - NonRecursiveClauses, - RecursiveClauses)
recursion(Proc,NonRCls,RCls):-
   recursive_clauses(Proc,NonRCls,RCls),
   \+ RCls = [],
   !.

% Separates the recursive and the nonrecursive clauses of a set of defining
% clauses.
recursive_clauses([],[],[]).
recursive_clauses([Clause-Ind|RProc],NonRCls,[Clause-Ind|RCls]):-
   recursive_clause(Clause-Ind),
   !,
   recursive_clauses(RProc,NonRCls,RCls).
recursive_clauses([Clause-Ind|RProc],[Clause-Ind|NonRCls],RCls):-
   recursive_clauses(RProc,NonRCls,RCls).

% Checks whether a clause is directly recursive.
recursive_clause([[PredX|Args]|Body]-_):-
   \+ \+ member([PredX|BodyArgs],Body),
   same_length(Args,BodyArgs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% If the set of bound arguments in ArgsOld is a subset of the set of bound
% arguments in ArgsNew, the value reorder is returned, otherwise the value
% no-reorder.  This predicate is used to determine whether a call to inverse for
% a literal should be performed in case a previous call was successful for the
% same literal.
```

```
reorder(ArgsOld,ArgsNew,reorder):-
   \+ ins_contained_in(ArgsOld,ArgsNew),
   !.
reorder(_,_,no-reorder).

% Retrieves the minimal degree of nondeterminacy (N) introduced by the nonunit
% clauses defining a literal by checking the failure-entries in the chart.
degree_proc([_-Ind|RProc],PredX,N,D):-
   check_chart(Ind,PredX,N1,D),
   failure_degree(RProc,PredX,N1,N,D).

% For all clauses in a procedure the chart is checked for failure-entries to
% determine the minimal degree of nondeterminacy introduced by the procedure:
% the minimal degree of nondeterminacy introduced is taken to be the smallest
% degree of nondeterminacy found in a failure-entry for one of the rhs-literals
% of the body of one of the clauses in the procedure for which no success-entry
% is found in the chart.  The indices in the chart-entries are used to make sure
% that that the right failure/success-entries are considered, namely only those
% that resulted from a call to inverse/3 instigated by one of the clauses in the
% procedure.
failure_degree([],_,N,N,_).
failure_degree([_-Ind|RProc],PredX,N0,N,D):-
   check_chart(Ind,PredX,N1,D),
   ( N1 < N0 ->
     N1 = N2
   ; N0 = N2
   ),
   failure_degree(RProc,PredX,N2,N,D).

% check_chart/4 retrieves the Degree of all entries indicating the failure of
% the inversion of a predicate instigated by clause Ind for which no
% success-entry was found.  This indicates there is no ordering of the body of
% clause Ind which allows a successful call to this predicate.
check_chart(Ind,PredX,N,D):-
   findall(
           Degree,
           (failure(Ind,_,Degree),
           \+ (success(Indices,[[[PredX|_]|_]-_|_]-_),member(Ind,Indices)),
           Degree > D),
           Degrees
           ),
   smallest(Degrees,N,D).

% Picks the smallest integer from a list of integers.  If Degrees equals the
% empty list D, i.e.  the current allowed degree of nondeterminacy, is returned.
smallest([],D,D).
smallest(Degrees,Smallest,_):-
```

```
   sort(Degrees,[Smallest|_]).

% Retrieves the actual degree of nondeterminacy (N) introduced by the unit
% clauses defining a literal.  In case there are no defining unit clauses
% degree_evid/2 returns the value '0'.
degree_evid(D,[_|Args],DefCl,N):-
   bound_positions(Args,1,Indices),
   projects(DefCl,Indices,RelArgsLL),
   !,
   non_determinate(RelArgsLL,D,N).

% This predicate takes care of fail anticipation:  if it fails a fail has been
% anticipated.
no_fail_anticipation([],[],_,_).
no_fail_anticipation(ArgsOld,ArgsNew,_,_):-
   \+ ins_contained_in(ArgsNew,ArgsOld).
no_fail_anticipation(ArgsOld,ArgsNew,MinimalDegree,AllowedDegree):-
   ins_contained_in(ArgsNew,ArgsOld),
   !,
   MinimalDegree =< AllowedDegree.

% ins_contained_in/2 is true if all arguments that are marked as bound in
% ArgsOld are marked as bound in ArgsNew as well:  if the set of bound arguments
% in ArgsNew is a subset of the set of bound arguments of ArgsOld.
% ins_contained_in( + ArgsNew, + ArgsOld)
ins_contained_in([],[]).
ins_contained_in([_-StatusOld|ArgsOld],[_-StatusNew|ArgsNew]):-
   ( StatusOld == in ->
     StatusNew == in,
     ins_contained_in(ArgsOld,ArgsNew)
   ; ins_contained_in(ArgsOld,ArgsNew)
   ).

% checks whether a set of bound arguments in ArgsNew is a proper subset of the
% set of bound arguments in ArgsOld.
ins_properly_contained_in(ArgsNew,ArgsOld):-
   ins_contained_in(ArgsNew,ArgsOld),
   \+ ins_contained_in(ArgsOld,ArgsNew).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% On the basis of BoundArgs the instantiated arguments of the OldRhs are marked
% as in.
bound_args_oldrhs([],_).
bound_args_oldrhs([[_PredX|Args]|Literals],BoundArgs):-
   bound_args(Args,BoundArgs),
   bound_args_oldrhs(Literals,BoundArgs).
```

```
bound_args([],_).
bound_args([Arg-Status|Args],BoundArgs):-
   var(Status),
   ( ground(Arg)
   ; contains_var(Arg,BoundArgs)
   ),
   !,
   Status = in,
   bound_args(Args,BoundArgs).
bound_args([_|Args],BoundArgs):-
   bound_args(Args,BoundArgs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The arguments of the literals in NewestRhs which become instantiated upon
% execution of these literals are determined by means of a call to status/1
% which computes the status of the arguments of all the literals in the grammar
% with the internal representation of the input-grammar in which the original
% body of a clause is substituted by the NewestRhs.  The code for status/1 is
% given in part C of the appendix.
bound_args_newrhs([],_,_,[],_,_):-
!.
bound_args_newrhs(_,Head,NewestRhs,NewBoundArgs,Ind,I):-
   copy_term(Head,HeadN),
   select(_-Ind,I,I1),
   !,
   keys_and_values(I1,I2,_),
   append([[HeadN|NewestRhs]],I2,I3),
   status(I3),
   member([_|NewestRhs],I3),
   status_args_rhs(NewestRhs,NewBoundArgs).

% The predicate status_args_rhs/2 is used to determine which arguments in a list
% of literals are 'in' or 'out'.
status_args_rhs(Rhs,Statuss):-
   status_args_rhs(Rhs,[],StatusL),
   ( \+ StatusL = [] ->
     setof(
          Arg-Status,
          (member(Arg-Status,StatusL)),
          Statuss
          )
   ; Statuss = []
   ).
status_args_rhs([],L,L).
status_args_rhs([[_PredX|Args]|Literals],Statuss,L):-
   include(status_in_out,Args,Statuss1),
   append(Statuss,Statuss1,L1),
```

```
   status_args_rhs(Literals,L1,L).

status_in_out(Arg):-
   ( status_in(Arg)
   ; status_out(Arg)
   ).

status_in(_-Status):-
   Status == in.
status_out(_-Status):-
   Status == out.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Selects a rhs-literal in OldRhs on the basis of the in-markings of the
% arguments in the literals.  Preference is given to the literal with the
% largest number of instantiated arguments, i.e.  arguments marked as in.  The
% output-argument OlderRhs is the list of remaining rhs-literals.
select_rhs_literal([Literal],Literal,[]):-
   !.
select_rhs_literal(OldRhs,SelLit,OlderRhs):-
   sort_by_in_args(OldRhs,SortedOldRhs),
   choose_literal(SortedOldRhs,SelLit,OlderRhs).

% Enumerates the literals of the ordered OldRhs ([First|Literals]).
choose_literal([Literal|Rest],Literal,Rest).
choose_literal([First|Literals],Literal,[First|SubRest]):-
   choose_literal(Literals,Literal,SubRest).

% Sorts the literals in OldRhs on the basis of the number of their instantiated
% arguments.
sort_by_in_args(Rhs,SortedRhs):-
   number_ins(Rhs,InsRhs),
   keysort(InsRhs,Sorted),
   keys_and_values(Sorted,_,StripSorted),
   rev(StripSorted,SortedRhs).

% Attaches an index to each literal in a list of literals which indicates the
% number of arguments of this literals that is instantiated.
number_ins([],[]).
number_ins([[PredX|Args]|Rhs],[InsLength-[PredX|Args]|R1]):-
   include(status_in,Args,InArgs),
   length(InArgs,InsLength),
   number_ins(Rhs,R1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% On the basis of the binding information associated with the arguments of a
% literal the maximal number of defining clauses with which this literal can
```

```
% unify is computed.
degree_of_nondeterminacy(D,_,DefCls):-
   length(DefCls,L),
   L =< D,
   !.
degree_of_nondeterminacy(D,[_|Args],DefCl):-
   bound_positions(Args,1,Indices),
   projects(DefCl,Indices,RelArgsLL),
   !,
   \+ non_determinate(RelArgsLL,D,_).

% Determines which argument positions of a literal are bound.
bound_positions([],_,[]).
bound_positions([_-Status|ArgsOld],N,Ns):-
   \+ Status == in,
   !,
   NN is N+1,
   bound_positions(ArgsOld,NN,Ns).
bound_positions([_|ArgsOld],N,[N|Ns]):-
   NN is N+1,
   bound_positions(ArgsOld,NN,Ns).

% This is an adaption of the Quintus-library predicate project/3 to deal with a
% list of Indices instead of a single Index (K). Collects the arguments of all
% the head-literals of the defining clauses which correspond to the instantiated
% arguments of a literal.  The output-argument is a list of lists containing the
% relevant arguments of each defining clause ([AllArgs|AllArgsRest]).
projects([[[_|Arguments]|_]-_|Facts],Inds,[AllArgs|AllArgsRest]) :-
   nths(Inds, Arguments, AllArgs),
   nths_args(Facts, Inds, AllArgsRest).

nths_args([], _, []).
nths_args([[[_|Arguments]|_]-_|Facts], Ks, [AllArgs|AllArgsRest]) :-
   nths(Ks, Arguments, AllArgs),
   nths_args(Facts, Ks, AllArgsRest).

nths([],_,[]).
nths([Ind|Inds], Fact, [Arg|Args]):-
   nth1(Ind, Fact, Arg),
   nths(Inds, Fact, Args).

% On the basis of the list of lists of relevant arguments the maximal degree of
% nondeterminacy a literal can introduce is determined:  for each member of L it
% is checked with how many other members of L it unifies.  This number is
% compared to Degree, i.e.  the admissible degree of nondeterminacy.
non_determinate([],_,-).
non_determinate(L,AllowedDegree,ActualDegree):-
```

```
   member(Member1,L),
   findall(
           Member2,
           (member(Member2,L),
           Member2 = Member1),
           UnifyingMembersL
           ),
   length(UnifyingMembersL,ActualDegree),
   ActualDegree > AllowedDegree.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% procedure/3 and evidence/3 are the predicates to retrieve the clauses defining
% a literal from the input-grammar.
procedure([PredX|Args],Procedure,I):-
   findall(
           [[PredX|Args1]|B]-N,
           (member([[PredX|Args1]|B]-N,I),
           same_length(Args,Args1)),
           Procedure
           ),
% At least one member of the retrieved procedure should have a non-empty body.
   member([_|NB]-_,Procedure),
   \+ NB = [].

evidence([PredX|Args],Evidence,I):-
   findall(
           [[PredX|Args1]|B]-N,
           (member([[PredX|Args1]|B]-N,I),
           B = [],
           same_length(Args,Args1)),
           Evidence
           ),
% If there are no unit clauses defining the literal evidence/3 fails.
   \+ Evidence = [].


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% After the clauses defining a literal have been retrieved from the internal
% representation of the input-grammar the binding information according to which
% they have to be inverted is added to the clauses by means of the
% status-indices associated to the arguments of the literals of these clauses.
% Before this can be done it is necessary to remove any bindings remaining from
% previous attempts to invert these clauses.
init_status_cls([],[]).
init_status_cls([FirstCl-Ind|RCls],[FirstClN-Ind|RClsN]):-
   init_status_cl(FirstCl,FirstClN),
   init_status_cls(RCls,RClsN).
```

```
init_status_cl([],[]).
init_status_cl([Literal|RLiterals],[LiteralN|RLiteralsN]):-
   init_status_literal(Literal,LiteralN),
   init_status_cl(RLiterals,RLiteralsN).

init_status_literal([],[]).
init_status_literal([PredX|Args],[PredX|ArgsN]):-
   init(Args,ArgsN).

init([F-_|R],[F-_|R1]):-
   init(R,R1).
init([],[]).

add_new_status([],[]).
add_new_status([Cl-Ind|Cls],[NewCl-Ind|NewCls]):-
   add_status_inds(Cl,NewCl),
   add_new_statuss(Cls,NewCls).

initialize([PredX|Args],[[Head|B]-Ind|RProc],Proc,I0,I):-
   init_status_cls([[Head|B]-Ind|RProc],Proc),
   init_input_markings_proc(Proc,I0,I1),
   mark_bound_args(Proc,[PredX|Args],I1,I).

init_input_markings_proc([],I,I).
init_input_markings_proc([_-Ind|RProc],I0,I):-
   member(Cl0-Ind,I0),
   init_status_cl(Cl0,Cl),
   select(Cl0-Ind,I0,I1),
   append(I1,[Cl-Ind],I2),
   init_input_markings_proc(RProc,I2,I).

mark_bound_args([],_,I,I).
mark_bound_args([[[PredX|Args1]|_]-Ind|RProc],[PredX|Args],I0,I):-
   select([[PredX|Args1]|B]-Ind,I0,I1),
   status_head(Args,Args1),
   append(I1,[[[PredX|Args1]|B]-Ind],I2),
   mark_bound_args(RProc,[PredX|Args],I2,I).

status_head([],[]).
status_head([_Arg-Status|Args],[_Arg1-Status1|Args1]):-
   ( Status == in ->
     Status1 = in
   ; true
   ),
   ( Status == out ->
     Status1 = out
```

```
   ; true
   ),
   status_head(Args,Args1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Finds a new predicate name given an old predicate name.
new_pred_name(Pred-N,NewPredName):-
   new_name(1,Pred-N,NewPredName).

new_name(C,PredName-N,PredName-NewN):-
   NewN is N+C,
   \+ success(_,[[PredName-NewN|_]-_|_]),
% Check the chart whether the new name is not already used.
   !.
new_name(C,PredName-N,PredName-NewN):-
   NewC is C+1,
   new_name(NewC,PredName-N,PredName-NewN).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% On the basis of a new predicate name new_name_proc/3 renames a procedure.  By
% means of a call to new_name_body/4 possible recursive rhs-literals are renamed
% as well.
new_name_proc([],_,[]).
new_name_proc([[[Pred-X|Args]|B]-Ind|RProc],Pred-Z,[[[Pred-Z|Args]|NB]-Ind|NRProc]):-
   new_name_body(B,NB,Pred-Z,Pred-X),
   new_name_proc(RProc,Pred-Z,NRProc).

new_name_body([],[],_,_).
new_name_body([[Pred-X|Args]|RB],[[Pred-Z|Args]|NRB],Pred-Z,Pred-X):-
   !,
new_name_body(RB,NRB,Pred-Z,Pred-X).
new_name_body([Lit|RB],[Lit|NRB],Pred-Z,Pred-X):-
   new_name_body(RB,NRB,Pred-Z,Pred-X).

% In case a recursive procedure has been reformulated renaming a procedure is a
% more complicated, because during reformulation type information is added to
% the predicate names.
new_name_ref_proc([],_,[]).
new_name_ref_proc([[[Pred-Type-X|Args]|B]-Ind|RProc],Pred-Z,
   [[[Pred-Type-Z|Args]|NB]-Ind|NRProc]):-
   new_name_ref_body(B,NB,Pred-Z,Pred-X),
   new_name_ref_proc(RProc,Pred-Z,NRProc).
new_name_ref_proc([[[Pred-X|Args]|B]-Ind|RProc],Pred-Z,[[[Pred-Z|Args]|NB]-Ind|NRProc]):-
   new_name_ref_body(B,NB,Pred-Z,Pred-X),
   new_name_ref_proc(RProc,Pred-Z,NRProc).

new_name_ref_body([],[],_,_).
```

```
new_name_ref_body([[Pred-Type-X|Args]|RB],[[Pred-Type-Z|Args]|NRB],Pred-Z,Pred-X):-
   new_name_ref_body(RB,NRB,Pred-Z,Pred-X).
new_name_ref_body([Lit|RB],[Lit|NRB],Pred-Z,Pred-X):-
   new_name_ref_body(RB,NRB,Pred-Z,Pred-X).
```

C BINDING STATUS COMPUTATION

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% status/1 determines the binding status of the arguments in the literals of the
% input-grammar (I) by means of calling the various predicates corresponding to
% the rules to determine which arguments in a literal are bound at the time of
% execution and which are bound after the computation is completed.  The status
% of an argument (‘in’, ‘out’ or ‘unknown’) is marked in the internal
% representation of the input-grammar by means of instantiating the
% status-index.  In order to avoid infinite loops while applying the rules to
% determine the status of an argument, an attempt is made to determine the
% status of an argument in a literal in the input-grammar by applying all the
% rules.  If this does not succeed the next argument is considered.  This goes on
% until the complete grammar is considered and then again all arguments in the
% grammar are considered.  Once no more arguments can be given a status
% according to the rules the in _out computation is finished.

% status( + InputGrammar).
status(I):-
   a_in(I),
   !,
   status(I).
status(I):-
   b_in(I),
   !,
   status(I).
status(I):-
   c_in(I),
   !,
   status(I).
status(I):-
   d_in(I),
   !,
   status(I).
status(I):-
   f_in(I),
   !,
   status(I).
status(I):-
   g_in(I),
   !,
   status(I).
```

```
status(I):-
   a_out(I),
   !,
   status(I).
status(I):-
   b_out(I),
   !,
   status(I).
status(I):-
   c_out(I),
   !,
   status(I).
status(_).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% These are the rules to compute the binding status of the arguments in the
% input-grammar, as proposed by Strzalkowski.
a_in(I):-
   member(Clause,I),
   member([_PredX|Args],Clause),
   member(Arg-Status,Args),
   var(Status),
   atomic(Arg),
   Status = in.
b_in(I):-
   member(Clause,I),
   member_prefix([PredX|Args],Clause,PreviousLiterals),
   select(Arg-Status,Args,Args1),
   var(Status),
   compound(Arg),
   ( ground(Arg)
   ; Arg =..  [_|ListArg],
     all_in(ListArg,[[PredX|Args1]|PreviousLiterals])
   ),
   Status = in.
c_in(I):-
   member([_Head|Body],I),
   member_prefix([_PredX|Args],Body,PreviousLiterals),
   member(Arg-Status,Args),
   var(Status),
   in_out_prev_lit(Arg,PreviousLiterals),
   Status = in.
d_in(I):-
   member([[_HP|HArgs]|Body],I),
   member([_PredX|Args],Body),
   member(Arg-Status,Args),
   var(Status),
```

```
   in_out_member(Arg-Status1,HArgs),
   Status1 == in,
   Status = in.
f_in(I):-
   member([[_PredX|Args]|_Body],I),
   select(Arg-Status,Args,Args1),
   var(Status),
   in_out_member(Arg-Status1,Args1),
   Status1 == in,
   Status = in.
g_in(I):-
   member([Head|_Body],I),
   member([PredX|Args],[Head]),
   nth1(N,Args,_Arg-Status),
   var(Status),
   findall(
           BodyIn,
           (member([[HeadPredX|_]|Body],I),
           member([BodyPredX|BodyArgs],Body),
           BodyPredX = PredX,
           same_length(BodyArgs,Args),
           HeadPredX \== BodyPredX,
           nth1(N,BodyArgs,_NthArg-BodyIn)),BodyIns
           ),
   BodyIns \== [],
   all_are_in(BodyIns),
   Status = in.

a_out(I):-
   member([_Head|Body],I),
   member_prefix([PredX|Args],Body,PreviousLiterals),
   select(Arg-Status,Args,Args1),
   var(Status),
   compound(Arg),
   Arg =..  [_|ListArg],
   one_out(ListArg,[[PredX|Args1]|PreviousLiterals]),
   Status = out.
b_out(I):-
   member([_H|Body],I),
   member([PredX|Args],Body),
   nth1(N,Args,_Arg-Status),
   var(Status),
   findall(
           HeadStatus,
           (member([[HeadPredX|HeadArgs]|_Body],I),
           HeadPredX = PredX,
           same_length(HeadArgs,Args),
```

```
          nth1(N,HeadArgs,_NthArg-HeadStatus)),
          HeadStatuss
          ),
   at_least_one_in(HeadStatuss),
   Status = out.
c_out(I):-
   member([[PredX|Args]|Body],I),
   member(Arg-Status,Args),
   var(Status),
   member([BodyPredX|BodyArgs],Body),
   in_out_member(Arg-Status1,BodyArgs),
   ( BodyPredX = PredX,same_length(BodyArgs,Args) ->
     Status = unknwn
   ; Status1 == out,
     Status = out
   ).

% One of the arguments in a list of arguments can be proven as having status
% 'out'
one_out([F|R],PrevLiterals):-
   one_out([F|R],PrevLiterals,StatusList),
   member(out,StatusList).
one_out([],_,_).
one_out([F|R],PrevLiterals,[B|T]):-
   atomic(F),
   !,
   B = in,
   one_out(R,PrevLiterals,T).
one_out([F|R],PrevLiterals,[B|T]):-
   var(F),
   !,
   ( in_prev_lit(F,PrevLiterals), B = in
   ; out_prev_lit(F,PrevLiterals), B = out
   ),
   one_out(R,PrevLiterals,T).
one_out([F|R],PrevLiterals,[B|T]):-
   F =..  [_|List],
   one_out(List,PrevLiterals,B),
   one_out(R,PrevLiterals,T).

% All arguments in a list of arguments are indexed as 'in'.
all_are_in([X|Y]):-
   X == in,
   all_are_in(Y).
all_are_in([]).

% All arguments in a list of arguments can be proven to be 'in'.
```

```
all_in([],_).
all_in([F|R],PrevLiterals):-
   atomic(F),
   !,
   all_in(R,PrevLiterals).
all_in([F|R],PrevLiterals):-
   var(F),
   !,
   in_prev_lit(F,PrevLiterals),
   all_in(R,PrevLiterals).
all_in([Arg|R],PrevLiterals):-
   Arg =..  [_|ListArg],
   all_in(ListArg,PrevLiterals),
   all_in(R,PrevLiterals).

% Var is 'in' or 'out' in a previous literal.
in_out_prev_lit(Var,PrevLits):-
   in_prev_lit(Var,PrevLits).
in_out_prev_lit(Var,PrevLits):-
   out_prev_lit(Var,PrevLits).

% Var is 'in' in a previous literal.
in_prev_lit(Var,[[_PredX|Args]|_Lits]):-
   in_out_member(Var-Status,Args),
   Status == in.
in_prev_lit(Var,[_F|R]):-
   in_prev_lit(Var,R).

% Var is 'out' in a previous literal.
out_prev_lit(Var,[[_PredX|Args]|_Lits]):-
   in_out_member(Var-Status,Args),
   Status == out.
out_prev_lit(Var,[_F|R]):-
   out_prev_lit(Var,R).

% At least one of the elements of a list of arguments is indexed as 'in'.
at_least_one_in([F|_R]):-
   var(F),
   !,
   fail.
at_least_one_in([unknwn|R]):-
   at_least_one_in(R).
at_least_one_in([in|R]):-
   ground(R).
at_least_one_in([out|R]):-
   ground(R).
```

D ARGUMENT SERIES COMPUTATION AND INTERPRETATION

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checks for all recursive clauses in a recursive proedure whether a bound
% peeling argument series can be found; in that case termination of the
% recursive procedure is ensured.
termination_ensured([],_).
termination_ensured([RCl-_|RCls],I):-
    in_peeling_series(RCl,I),
    termination_ensured(RCls,I).

% Computes all argument series corresponding to the bound argument positions and
% subsequently checks whether they are peeling.  During the computation of an
% argument series and its corresponding auxiliary argument series the relation,
% between the individual arguments in an argument series, referred to as
% 'order', is stored:  the order between two arguments X and Y in an argument
% series is either peeling (represented as 'p'), in which case contains_var(X,Y)
% holds or building (represented as 'b'), in which case contains_var(Y,X) holds.
% In case two arguments in an argument series are identical this is represented
% by the order '-'.  Upon the computation of a direct argument series a list of
% orders is returned.  This list is referred to as the 'orders' of an argument
% series.  peeling_series/2 checks the orders of an argument series to determine
% whether the argument series computed is peeling.
in_peeling_series([[PredX|Args]|Body],I):-
    bound_positions(Args,1,BNs),
    all_n_series(BNs,[[PredX|Args]|Body],Orders,I),
    peeling_series(Orders,_).

% In case one of the argument series for a bound argument position is a peeling
% argument series peeling_series/2 is true.
peeling_series([O|_],true):-
    \+ O == [],
    p_series(O),
    !.
peeling_series([_|Os],Result):-
    peeling_series(Os,Result).

% Inspects the orders of a single argument series.  In order for the argument
% series to be peeling all must be peeling.
p_series([]).
p_series([O|Os]):-
    member(p,O),
    p_series(Os).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Finds all argument series (and all the corresponding auxiliary argument
```

```
% series) for a list of argument-positions and returns their orders.
all_n_series([],_,[],_).
all_n_series([N|Ns],RCl,[Order|Orders],I):-
    series(RCl,N,Order,I),
    all_n_series(Ns,RCl,Orders,I).

% N indicates for which argument position in the head-literal of the recursive
% clause we are looking for the argument series.  The arguments of the literals
% in the recursive clause are extended with two additional indices to indicate
% in which literal in the clause (L) and at which position in the literal they
% occur (A). For example:  Arg-L-A, indicates that Arg is the Ath argument of
% rhs-literal L. (The head-literal is L = 0.)  series/4 is thus called with as
% its second argument the Nth argument of the head-literal of the recursive
% clause.  Recall that the arguments in a literal already had a status-index.
% In total each argument has three indices now.  findall/3 is used to ensure that
% all argument series are found.
series([[PredX|Args]|Body],N,OrderList,I):-
    nth1(N,Args,ArgY-_Status),
    findall(
            Order,
            x_argument([[PredX|Args]|Body],ArgY-0-N,Order,I,N),
            OrderList
            ).

% x_argument/5 given an argument of the head-literal of the recursive clause
% tries to find the next argument in the argument series.  At the moment the next
% argument in the argument series is an argument of the recursive rhs-literal,
% it is checked whether the argument positions of the beginning and the end of
% the argument series are identical.  If so an argument series is found and the
% corresponding orders are percolated upward.  If this is not the case the next
% argument in the argument series is sought in the participating rhs-literal by
% means of calling y_argument/5 and the corresponding auxiliary argument series
% is computed, followed by a recursive call to x_argument/5.
x_argument([Head|Body],ArgY-L-_,[O|Os],I,N):-
    nth1(L1,Body,[BodyPredX|BodyArgs]),
    L1 > L,
    nth1(A1,BodyArgs,ArgX-_Status),
    contains(ArgY,ArgX,O),
    ( recursive_clause([Head|[[BodyPredX|BodyArgs]]]-_),
      A1 = N,
      !,
      Os = []
    ; y_argument([Head|Body],ArgX-L1-A1,ArgY2-L1-A2,SubOrderList,I),
      x_argument([Head|Body],ArgY2-L1-A2,RestOs,I,N),
      append(SubOrderList,RestOs,Os)
    ).
```

```
% y_argument/5 tries to connect ArgX-L1-A1 to another argument in the same
% participating rhs-literal.  This is possible only in case, it is possible to
% prove an auxiliary argument series between these two arguments.
y_argument([Head|Body],_ArgX-L1-A1,ArgY-L1-A2,SubOrderList,I):-
    nth1(L1,Body,[_BodyPredX|BodyArgs]),
    nth1(A2,BodyArgs,ArgY-_Status),
    A1 =\= A2,
    auxiliary_series([Head|Body],L1,A1,A2,SubOrderList,I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% A defining clause is retrieved from the input-grammar and it is tried whether
% an auxiliary argument series can be proven.
auxiliary_series([_|Body],L,A1,A2,SubOrder,I):-
    nth1(L,Body,[BodyPredX|BodyArgs]),
    defining_clause([BodyPredX|BodyArgs],[[DefClPred|DefClArgs]|DefClBody],I,INew),
    nth1(A1,DefClArgs,ArgX-_Status1),
    nth1(A2,DefClArgs,ArgY-_Status2),
    aux_series([[DefClPred|DefClArgs]|DefClBody],ArgX-0-A1,ArgY-0-A2,SubOrder,INew).

% First it is checked whether the auxiliary argument series runs in the
% head-literal of the defining clause only.  If this is not the case aux/6 is
% called to find the next argument in the argument series.
aux_series([Head|Body],ArgX-0-A1,ArgY-0-A2,AuxOrder,I):-
    ( contains(ArgX,ArgY,O) ->
      AuxOrder = [O]
    ; aux([Head|Body],ArgX-0-A1,AuxOrder,I,ArgY,A2)
    ).

% A rhs-literal from the defining clause is picked and two arguments in it which
% stand in the contains-relation.  Then it is tried to combine the last argument
% in the argument the relevant argument of the head-literal.  If this is not
% possible a pair of arguments in another rhs-literal is chosen by means of a
% recursive call to aux/6.  Note that in contrast to an argument series, the
% order of the rhs-literals in a clause in which an auxiliary argument series is
% proven is unimportant.
aux([Head|Body],ArgX-L-_A,Order,I,HeadArgY,HeadPos):-
    nth1(L1,Body,[_BodyPredX|BodyArgs]),
    \+ L1 = L,
    nth1(A1,BodyArgs,ArgY-_Status),
    contains(ArgX,ArgY,O),
    y_argument([Head|Body],ArgY-L1-A1,ArgX2-L1-A2,SubOrderList,I),
    ( contains(ArgX2,HeadArgY,O1),
      append([O],SubOrderList,Ox),
      append(Ox,[O1],Order)
    ; aux([Head|Body],ArgX2-L1-A2,Os,I,HeadArgY,HeadPos),
      append([O],SubOrderList,Ox),
      append(Ox,Os,Order)
```

```
    ).

% By means of contains_var/2 the order between two arguments in an (auxiliary)
% argument series is determined.
contains(ArgY,ArgX,Order):-
    ( ArgX == ArgY ->
      Order = -
    ; contains_var(ArgX,ArgY) ->
      Order = p
    ; contains_var(ArgY,ArgX) ->
      Order = b
    ).

% Retrieves one of the defining clauses for the literal [PredX|Args] from the
% input-grammar in order to check whether an auxiliary argument series can be
% found in this clause.  This defining clause is removed from the input-grammar
% in order to guarantee the termination of the auxiliary argument series
% computation.
defining_clause([PredX|Args],[[PredX|Args1]|Body],I0,I):-
    select([[PredX|Args1]|Body]-_,I0,I),
    same_length(Args,Args1).
```

E ARGUMENT SERIES REVERSAL

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% In case given the binding information associated to a call to a recursive
% procedure no bound peeling argument series exists (termination_ensured/2 failed
% for the call), building_series_reverse/4 reverses a building argument series in
% all recursive clauses in the procedure.  Upon the reversal of the argument
% series the recursive procedure is immediatly reformulated by means of a call to
% reform/4.  The predicate reform/4 is given in appendix F.
% outs_base_case_call/4 determines which argument positions are instantiated upon
% the execution of the isolated base cases of the recursive procedure (OutNs).
% These are argument positions in the recursive clauses that are checked for a
% building argument series.  If an argument of the call to the isolated base cases
% of the recursive procedure does not become instantiated, it is of no use to
% invert building series to the corresponding argument position, because there
% will be no recursion controlling information available to make reformulation
% possible.  keys_and_values/3 is used to undo the clauses in the recursive
% procedure from their clause-indices.

% building_series_reverse( + IndexedNonRecursiveClauses,
% + IndexedRecursiveClauses, - ReformulatedRecursiveProcedure, + InputGrammar)
building_series_reverse(NonRClsInd,RClsInd,NProcs,I):-
   outs_base_case_call(NonRClsInd,RClsInd,OutNs,I),
   keys_and_values(NonRClsInd,NonRCls,_),
   keys_and_values(RClsInd,RCls,_),
```

```
   relevant_series_reverse(RCls,OutNs,N,I,NewRCls),
   reform(NewRCls,NonRCls,N,NProcs,I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Computes which arguments become instantiated upon the execution of the base
% case of a recursive procedure.
outs_base_case_call(NonRCls,RCls,OutNs,I):-
   select_rclss(RCls,I,I1),
   !,
   keys_and_values(I1,I2,_),
   status(I2),
   outs_base_cases(NonRCls,Outs,I2),
   intersection(Outs,OutNs).

% Removes a list of clauses from the internal representation of the input-grammar.
select_rclss([],I,I).
select_rclss([_-Ind|RCls],I0,I):-
   select(_-Ind,I0,I1),
   select_rclss(RCls,I1,I).

outs_base_cases([],[],_).
outs_base_cases([[[PredX|Args]|Body]-_|NonRCls],[Outs|ROuts],I):-
   member([[PredX|Args]|Body],I),
   outs(Args,Outs,1),
   outs_base_cases(NonRCls,ROuts,I).

% From a list of arguments the position of those are isolated which have
% as status index 'in' or 'out'.
outs([],[],_).
outs([_-Status|RArgs],[N|Ns],N):-
   ( Status == out
   ; Status == in
   ),
   !,
   NN is N+1,
   outs(RArgs,Ns,NN).
outs([_|RArgs],Ns,N):-
   NN is N+1,
   outs(RArgs,Ns,NN).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% In case for all recursive clauses in a procedure a building argument series at
% the same argument position can be reversed relevant_series_reverse/5 succeeds.
relevant_series_reverse(RCls,[UsedN|_Ns],UsedN,I,NewRCls):-
   all_n_series_reverse(RCls,UsedN,I,NewRCls),
   !.
relevant_series_reverse(RCls,[UsedN|Ns],UsedN,I,NewRCls):-
```

```
   relevant_series_reverse(RCls,Ns,UsedN,I,NewRCls).

% Reverses a building argument series at argument position N for all recursive
% clauses.
all_n_series_reverse([],_,_,[]).
all_n_series_reverse([RCl|RCls],N,I,[NewRCl|NewRCls]):-
   series_reverse(RCl,N,NewRCl,Result,I),
   Result == true,
% Result = true indicates that a building argument series has been inverted.
   all_n_series_reverse(RCls,N,I,NewRCls).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The difference between the following predicates and the predicates with the
% same predicate name minus the suffix _reverse given in the previous appendix
% is that an accumulator pair is passed along with the recursive clause to
% immediately reverse a building argument series in a recursive clause, when it
% is found:  NewRCl0 initially carries the original recursive clause, NewRCl upon
% the execution of x_argument_reverse/7 carries the new recursive clause with
% the reversed building argument series.  A third argument Result is used to
% determine whether a building argument series is reversed.
series_reverse([[PredX|Args]|Body],N,[[PredX|A]|B],Result,I):-
   nth1(N,Args,ArgY-_Status),
   x_argument_reverse([[PredX|Args]|Body],ArgY-0-N,N,[[PredX|Args]|Body],
   [[PredX|A]|B],Result,I).

x_argument_reverse([Head|Body],ArgY-L-A,N,NewRCl0,NewRCl,Result,I):-
   nth1(L1,Body,[BodyPredX|BodyArgs]),
   L1 > L,
   nth1(A1,BodyArgs,ArgX-_Status),
   contains(ArgY,ArgX,O),
   swap_args(O,ArgY-_-L-A,ArgX-_-L1-A1,NewRCl0,NewRCl1,Result),
   ( (recursive_clause([Head|[[BodyPredX|BodyArgs]]]-_),
     A1 = N) ->
     NewRCl1 = NewRCl
   ; y_argument_reverse([Head|Body],ArgX-L1-A1,ArgY2-L1-A2,NewRCl1,NewRCl2,Result,I),
     x_argument_reverse([Head|Body],ArgY2-L1-A2,N,NewRCl2,NewRCl,Result,I)
   ).

y_argument_reverse([Head|Body],ArgX-L1-A1,ArgY-L1-A2,NewRCl0,NewRCl,Result,I):-
   nth1(L1,Body,[_BodyPredX|BodyArgs]),
   nth1(A2,BodyArgs,ArgY-_Status),
   A1 =\= A2,
   findall(
           AuxOrder,
           auxiliary_series([Head|Body],L1,A1,A2,AuxOrder,I),
           AuxOrderList
           ),
```

```
% If all auxiliary argument series between A1 and A2 in literal L1 are building,
% A1 and A2 have to be swapped.
   ( all_building(AuxOrderList) ->
     swap_args(b,ArgX-_-L1-A1,ArgY-_-L1-A2,NewRCl0,NewRCl,Result)
   ; NewRCl0 = NewRCl
   ).

% Make sure that all auxiliary argument series found are building.  If not all
% auxiliary argument series are building or there are no auxiliary argument
% series all_building/1 fail.
all_building([]).
all_building([H|T]):-
   member(b,H),
   all_building(T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Upon execution of swap_args/6, NewRCl equals NewRCl0 except that the Ath
% argument of literal L of NewRCl0 has been interchanged with the A1th argument
% of literal L1 of NewRCl0.  In case swap_args/6 actually swapped arguments,
% Result is set to true.  It is necessary to do this to be able to determine
% whether it has been possible to reverse a buiding argument series.
swap_args(b,ArgY-_-L-A,ArgX-_-L1-A1,NewRCl0,NewRCl,true):-
   !,
   swap(NewRCl0,0,ArgY-_-L-A,ArgX-_-L1-A1,NewRCl).
swap_args(_,_,_,N,N,_).

swap([],_,_,_,[]).
swap([H0|T0],X,ArgY-_-X-A,ArgX-_-X-A1,[H|T]):-
% The arguments that have to be swapped occur in the same rhs-literal.
   !,
   s(0,A,ArgX-_,H0,H1),
   s(0,A1,ArgY-_,H1,H),
   XNew is X + 1,
   swap(T0,XNew,ArgY-_-X-A,ArgX-_-X-A1,T).
swap([H0|T0],X,ArgY-_-X-A,ArgX-_-L1-A1,[H|T]):-
% The arguments to be swapped occur in two different rhs-literals.
   !,
   s(0,A,ArgX-_,H0,H),
   XNew is X + 1,
   swap(T0,XNew,ArgY-_-X-A,ArgX-_-L1-A1,T).
swap([H0|T0],X,ArgY-_-L-A,ArgX-_-X-A1,[H|T]):-
% The arguments to be swapped occur in two different rhs-literals.
   !,
   s(0,A1,ArgY-_,H0,H),
   XNew is X + 1,
   swap(T0,XNew,ArgY-_-L-A,ArgX-_-X-A1,T).
swap([H0|T0],X,ArgY-_-L-A,ArgX-_-L1-A1,[H0|T]):-
```

```
% No arguments need to be swapped in this literal.
   XNew is X + 1,
   swap(T0,XNew,ArgY-_-L-A,ArgX-_-L1-A1,T).

% NArg replaces the Xth argument in the list of arguments of a literal
s(X,X,NArg,[_|Args],[NArg|Args]):-
   !.
s(X,A,NArg,[Arg|Args],[Arg|NArgs]):-
   XNew is X + 1,
   s(XNew,A,NArg,Args,NArgs).
```

F REFORMULATION

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% reform/4 is called to reformulate a recursive procedure such that a reversed
% building argument series can be used to control the recursion without changing
% the semantics of the input-grammar.  The reformulation performed by reform/4
% consists of three steps:
% 1.  a new intermediate clause is created by means of a call to
% intermediate_clause/3.
% 2.  the nonrecursive clauses of the recursive procedure are isolated by means
% of a call to isolated_base_cases/2.
% 3.  The recursive procedure is adapted by adding an additional context argument
% and changing the original nonrecursive clauses by means of a call to
% recproc_reform/3.
% The recursive clauses with which reform/4 is called are already modified in
% the sense that building argument series have been reversed.  In order to
% facilitate the treatment of the reformulated procedure in the overall
% inversion process the clauses resulting from reformulation get dummy
% clause-indices by means of a call to keys_and_values/3.


% reform( + RecursiveClauses, + NonRecursiveClauses,
% + ArgumentPositionReversedSeries, - IndexedNewProcedure)
reform(RCls,NRCls,N,IndexedNewProcs,I):-
   recproc_reform(RCls,NRCls,RefProc),
   new_clauses(RCls,NRCls,N,NewProc,I),
   append(NewProc,RefProc,NewProcs),
   keys_and_values(IndexedNewProcs,NewProcs,_).

% Creates a new intermediate clause and isolates the base cases of the recursive
% procedure.
new_clauses(RCls,[[Head|Body]|NRCls],N,[IntCl|IsoBCs],I):-
   intermediate_clause(RCls,Head,IntCl,N,I),
   isolated_base_cases([[Head|Body]|NRCls],IsoBCs).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The nonrecursive clauses of the recursive procedure are isolated by extending
% their predicate name with the type-information _base.
isolated_base_cases([],[]).
isolated_base_cases([[[Pred-X|Args]|Body]|NRCls],[[[Pred-base-X|Args]|Body]|IsoBCs]):-
   isolated_base_cases(NRCls,IsoBCs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% intermediate_clause/4 creates a new clause and then determines on the basis of
% the binding of the arguments of the original call to the recursive procedure
% and the output-arguments of the base case call which arguments should be
% shared in the new clause.
intermediate_clause(RCls,[Pred-X|Args],[[Pred-X|NewArgs],[Pred-base-X|BaseArgs],
   [Pred-rec-X|RecArgs]],N,I):-
   new_arguments(Args,NewArgs),
   base_call_args(RCls,NewArgs,BaseArgs,I),
   rec_call_args(NewArgs,N,BaseArgs,RecArgs).

new_arguments(Args,NewArgs):-
   new_args(Args,[],NewArgs).

new_args([],NewArgs,NewArgs).
new_args([_-Status|Args],NewArgs0,NewArgs):-
   append(NewArgs0,[_-Status],NewArgs1),
   new_args(Args,NewArgs1,NewArgs).

% base_call_args/2 ensures the sharing of the instantiated arguments of the
% original call to the recursive procedure with the arguments of base case call.
base_call_args(RCls,OAs,BAs,I):-
   copy_term(OAs,NOAs),
   all_contextual_ins(RCls,1,OAs,NOAs,BAs,I).

all_contextual_ins(_,_,[],[],[],_).
all_contextual_ins(RCls,N,[OA-Status|OAs],[_-Status|NOAs],[OA-Status|BAs],I):-
   Status == in,
   contextual_argument(RCls,N,I),
   !,
   NNew is N +1,
   all_contextual_ins(RCls,NNew,OAs,NOAs,BAs,I).
all_contextual_ins(RCls,N,[_-_|OAs],[NOA-_|NOAs],[NOA-_|BAs],I):-
   NNew is N + 1,
   all_contextual_ins(RCls,NNew,OAs,NOAs,BAs,I).

% Checks whether an argument is contextual in each recursive clause in a list of
% recursive clauses
contextual_argument([],_,_).
contextual_argument([RCl|RCls],N,I):-
```

```
   series(RCl,N,Order,I),
% If one of the argument series found is building or peeling the argument
% corresponding to argument position N is not contextual.
   \+ Order = [],
   contextual(Order),
   contextual_argument(RCls,N,I).

contextual([]).
contextual([O|Os]):-
   \+ member(b,O),
   \+ member(p,O),
   contextual(Os).

% rec_call_args/4 ensures that the recursion controlling information resulting
% from the base case call is fed into the call to the further adapted recursive
% procedure and that the instantiated arguments of the original call to the
% recursive procedure are used to constraint the recursion appropriately.
% Furthermore, it takes care of passing along the right arguments with the extra
% bundle-argument of the adapted recursive procedure:  the remaining
% output-arguments of the base case call and the remaining input-arguments of
% the original call to the recursive procedure.
rec_call_args(OAs,N,BAs,RAs):-
   all_outs(1,BAs,OAs,N,RAs1),
   bundle(BAs,1,OAs,N,RelOArgs),
   add_status([_|NArgs],[_|RelOArgs]),
   append(RAs1,[NArgs-_],RAs).

bundle([],_,[],_,[]).
bundle([BA|BAs],X,[_|OAs],N,[BA|NAs]):-
   \+ X == N,
   !,
   XNew is X+1,
   bundle(BAs,XNew,OAs,N,NAs).
bundle([_|BAs],_,[OA|_],_,[OA|BAs]).

all_outs(X,[_|BAs],[FirstRAO|FirstRAsO],N,[FirstRAO|FirstRAs]):-
   \+ X == N,
   !,
   XNew is X + 1,
   all_outs(XNew,BAs,FirstRAsO,N,FirstRAs).
all_outs(_,[BA|_],[_|FirstRAsO],_,[BA|FirstRAsO]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% recproc_reform/3 takes care of the further adaption of the recursive
% procedure.  The predicate name of the recursive procedure is extended with the
% type-information _rec to enhance the readability of the reformulated
% procedure.
```

```
recproc_reform(RCls,BaseCases,RefProc):-
   rec_reform(RCls,NewRCls),
   bas_reform(BaseCases,NewBaseCases),
   append(NewRCls,NewBaseCases,RefProc).

% All recursive clauses of the recursive procedure get an extra context argument
% Bundle in the head-literal and the recursive rhs-literal.
rec_reform([],[]).
rec_reform([[[Pred-X|Args]|Body]|RCs],[[[Pred-rec-X|NArgs]|NBody]|NRCs]):-
   append(Args,[BundleArg-_],NArgs),
   body_reform(Body,NBody,[Pred-X|Args],[BundleArg-_]),
   rec_reform(RCs,NRCs).

body_reform([],[],_,_).
body_reform([[Pred-X|Args2]|RhsLts],[[Pred-rec-X|NArgs]|NRhsLts],[Pred-X|Args1],
   [BundleArg-_]):-
   !,
   same_length(Args1,Args2),
   append(Args2,[BundleArg-_],NArgs),
   body_reform(RhsLts,NRhsLts,[Pred-X|Args1],BundleArg).
body_reform([[Pred2X2|Args2]|Lts],[[Pred2X2|Args2]|NLts],[PredX|Args1],BundleArg):-
   body_reform(Lts,NLts,[PredX|Args1],BundleArg).

% The original base cases of the recursive procedure are modified such that the
% extra bundle-argument is unfolded and the proper identification with the
% original arguments of these clauses is enforced.
bas_reform([],[]).
bas_reform([[[Pred-X|Args]|_]|BCs],[[[Pred-rec-X|NNArgs]]|NBCs]):-
   add_status([Pred-X|NArgs],[Pred-X|Args]),
   append(Args,[NArgs-_],NNArgs),
   bas_reform(BCs,NBCs).
```

References

[1] Beeri, C. & Ramakrishnan, R. [1991] On the Power of Magic. *Journal of Logic Programming 10(2).*

[2] Dymetman, M. & Isabelle, P. [1988] Reversible Logic Grammars for Machine Translation. In *Proceedings of th 2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*, Pittsburgh, PA.

[3] Dymetman, M., Isabelle, P. & Perrault, F. [1990] A Symmetrical Approach to Parsing and Generation. In *Proceedings of COLING-90*, Helsinki, Finland.

[4] Dymetman, M. [1992] A Generalized Greibach Normal Form for Definite Clause Grammars. In *Proceedings of COLING-90*, Nantes, France.

[5] Gerdemann, D. [1991] *Parsing and Generation of Unification Grammars.* PhD dissertation, Published as Beckman Institute Cognitive Science Technical Report CS-91-06, University of Illinois.

[6] Gerdemann, D. & Hinrichs, E. [1990] Functor-Driven Natural Language Generation with Categorial-Unification Grammar. In *Proceedings of COLING-90*, Helsinki, Finland.

[7] Martinovic, M. & Strzalkowski, T. [1992] Comparing Two Grammar-based Generation Algorithms: a case study. In *Proceedings of the 30th ACL*, Newark, DE.

[8] Peng, P. & Strzalkowski, T. [1990] An Implementation of a Reversible Grammar. In *Proceedings of the 8th Canadian Conference on Artificial Intelligence*, Ottawa, Canada.

[9] Pereira, F. & Shieber, S. [1987] *Prolog and Natural Language Analysis.* Volume 10 of CSLI Lecture Notes, Stanford, CA.

[10] O'Keefe, R. [1990] *The Craft of Prolog.* MIT Press, Cambridge, Massachusetts.

[11] Sterling, L. & Shapiro, E. [1986] *The Art of Prolog.* MIT Press, Cambridge, Massachusetts.

[12] Shieber, S., Van Noord, G., Pereira, F. & Moore, R. [1990] Semantic Head-Driven Generation. *Computational Linguistics 16(1).*

[13] Shoham, Y. & McDermott, D. [1984] Directed Relations and Inversion of Prolog Programs. In *Proceedings of the International Conference of Fifth Generation Computer Systems*, Institute for New Generation Computer Technology, Tokyo, Japan.

[14] Strzalkowski, T. [1990a] How to Invert a Parser into an Efficient Generator: an algorithm for logic grammars. In *Proceedings of COLING-90*, Helsinki, Finland.

[15] Strzalkowski, T. [1990b] Reversible Logic Grammars for Natural Language Parsing and Generation. *Computational Intelligence, 6(3).*

[16] Strzalkowski, T. [1991] A General Computational Method for Grammar Inversion. In *Proceedings Reversible Grammar Workshop*, Berkeley, CA.

[17] Strzalkowski T. & Peng, P. [1990] Automated Inversion of Logic Grammars for Generation. In *Proceedings of the 28th ACL*, Pittsburgh, PA.

[18] Ullman, J. [1985] Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems 10(3).*