A System for Transforming Logic Programs

Sophie Renault

Università degli Studi di Roma Tor Vergata
Dipartimento di Informatica, Sistemi e Produzione
Via della Ricerca Scientifica, 00133 Roma, Italy
e-mail: renault@iasi.rm.cnr.it

Abstract

We describe a system for transforming logic programs using the unfold/fold methodology. The system provides the user with some predefined transformation rules. A transformation process consists of a sequence of applications of those rules, starting from a given initial program. The user selects a transformation rule to be applied, which transforms the initial program into a new equivalent one. Several such derivation steps may be performed interactively, until the desired program is derived. Some strategies are also available, which perform fully automatically a sequence of derivation steps for realizing specific transformations.

The report is intended to describe the functionalities of the system, and it should serve as a user's manual.

Contents

1 Introduction **3**

2 An example transformation: The scalar product **3**

3 Transformations rules **5**
3.1 Rules 6
3.2 Semantics preservation 7

4 Representation of the transformation process **7**
4.1 Transformation tree 7
4.2 Internal representation of data structures 9

5 Running the transformer **11**
5.1 Installation 11
5.2 Invoking the transformer 12

6 Top-level commands **12**
6.1 Transformation rules 12
6.2 Other commands 19

7 Transformation strategies **25**
7.1 A strategy for removing unnecessary variables 25
7.2 A strategy for loop absorption 27
7.3 A strategy for reducing nondeterminism 28

8 Conclusions and future work **40**

9 Acknowledgments **40**

A Guidelines to the use of the transformer **41**

B Summary of help messages associated with top-level commands **44**

C Code of the Transformer **51**
C.1 `top_level.pl` 53
C.2 `definition_rules.pl` 65
C.3 `unfolding_rules.pl` 66
C.4 `folding_rules.pl` 69
C.5 `goal_replacement_rules.pl` 74
C.6 `clause_replacement_rules.pl` 76
C.7 `input_output.pl` 79
C.8 `msg_user.pl` 86
C.9 `struct.pl` 97
C.10 `basic_op.pl` 102
C.11 `environment.pl` 112
C.12 `uve.pl` 115
C.13 `lap.pl` 119
C.14 `determ_prog.pl` 137
C.15 `case_split_for_determ.pl` 144
C.16 `determ_prog_util.pl` 151
C.17 `determ_strat_parameters.pl` 155

References **158**

1 Introduction

We describe a system for transforming logic programs [6], based on the unfold/fold approach [3, 8, 13], which has been developed at Rome Tor Vergata University and IASI Institute of the National Research Council, Rome, Italy. Several people have been involved in this development, O. Aioni [1] has implemented the kernel on top of which some extensions have been done by E. Capogna, M. Gaspari [5], T. Morgante [7] and myself. The implementation has been done in Prolog. When using our tool, the user first gives an initial program that he wants to transform, together with a theory, which provides some lemmas which can be used during program transformation. A transformation consists of a sequence of applications of some transformation rules starting from the initial program. The application of a transformation rule on a program P_i transforms program P_i into a new program P_{i+1}, and thus, a transformation can be seen as a sequence of programs $P_1, \ldots, P_n$, that is called a *transformation sequence*.
A transformation sequence is constructed in one of the following two modes:

- interactive mode: the user selects a rule and some parameters, if necessary. For instance, he may select the *unfolding rule*, and a clause and an atom of that clause w.r.t. the clause should be applied. The selected rule is applied, if the conditions for its application w.r.t. the current program are satisfied. Thus, the current program is transformed into a new one, which becomes the new current one. The available transformation rules are presented in Section 3.
- automatic mode: the user selects a strategy which will automatically control the sequencing of the transformation rules to achieve some specific objectives. For instance, he may select the strategy for removing the unnecessary variables [11] occurring in the initial program. Thus, the the initial program is transformed, if the strategy terminates, into an equivalent program without unnecessary variables, by an automatic selection of the transformation rules at each step of the derivation. There are three strategies available, which are described in Section 7.

The system keeps track of the history of the transformation process so that it is possible to backtrack to some previous programs of the transformation sequence and explore alternative derivations. The user may also save and restore the history of a transformation. Some facilities are provided to ease the user's task in the interactive mode.

2 An example transformation: The scalar product

In this section we illustrate the use of our transformer. We borrow the following example from Burstall and Darlington [3]. This example is also used by M. Feather [4] to illustrate the ZAP system. Let x and y be two vectors of dimension n. The scalar product of x and y denoted by '$\cdot$' is defined by the formula:

$$x \cdot y = \Sigma_{i=1}^{n} x_i \ y_i$$

which can be computed by the following logic program (we omit the clauses for *mult* which are not important for our purposes here):

1 $s\_p([], [], 0, 0) \leftarrow$
2 $s\_p([A|B], [C|D], s(N), X) \leftarrow s\_p(B, D, N, Y),\ mult(A, C, Z),\ plus(Y, Z, X)$
3 $plus(0, A, A) \leftarrow$
4 $plus(s(A), B, s(C)) \leftarrow plus(A, B, C)$

Now suppose that we want a program which computes, given the vectors a, b, c, and d, the formula $a \cdot b + c \cdot d$. Such a program can be obtained by introducing the clause:

5 $p(A, B, C, D, N, X) \leftarrow s\_p(A, B, N, Y),\ s\_p(C, D, N, Z),\ plus(Y, Z, X)$

The efficiency of the program consisting of clauses 1, 2, 3, 4, and 5 can be improved by avoiding the two separate calls to s_p in clause 5. Below we show how to derive from clauses 1, 2, 3, 4,

and 5, an equivalent program which achieves that goal, by applying the *unfolding*, *folding*, and *goal replacement* transformation rules (see Section 3 for a formal presentation of those rules). We show a session of our system corresponding to this derivation. For the sake of readibility, all parts of the session typed by the user are underlined, and cpmmands are typed entirely, although they can be abbreviated (see Section 6.
The initial program, made out of clauses 1, 2, 3, and 4, is available in a file named `scalar_product`. It is selected by applying the command `start`. Clause 5 is introduced by applying the command `ts_define`, which correspond to the *definition introduction* transformation rule.

```
| ?- start(scalar_product).

Initial program:
1. s_p([ ],[ ],0,0).
2. s_p([A|B],[C|D],s(E),F) :- s_p(B,D,E,G), mult(A,C,H), plus(H,G,F).
3. plus(0,A,A).
4. plus(s(A),B,s(C)) :- plus(A,B,C).

| ?- ts_define.
enter definition clause:
|:  p(A,B,C,D,N,X) :- s_p(A,B,N,X1), s_p(C,D,N,X2), plus(X1,X2,X).

5. p(A,B,C,D,E,F) :- s_p(A,B,E,G), s_p(C,D,E,H), plus(G,H,F).

| ?- unfold(5,1).

by unfolding clause 5 w.r.t. atom 1 we get:
6. p([ ],[ ],A,B,0,C) :- s_p(A,B,0,D), plus(0,D,C).
7. p([A|B],[C|D],E,F,s(G),H) :- s_p(B,D,G,I), mult(A,C,J), plus(J,I,K),
          s_p(E,F,s(G),L), plus(K,L,H).

| ?- unfold(6,1).

by unfolding clause 6 w.r.t. atom 1 we get:
8. p([ ],[ ],[ ],[ ],0,A) :- plus(0,0,A).

| ?- unfold(8,1).

by unfolding clause 8 w.r.t. atom 1 we get:
9. p([ ],[ ],[ ],[ ],0,0).

| ?- unfold(7,4).

by unfolding clause 7 w.r.t. atom 4 we get:
10. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- s_p(B,D,I,K), mult(A,C,L),
          plus(L,K,M), s_p(F,H,I,N), mult(E,G,O), plus(O,N,P), plus(M,P,J).

| ?- add_law.
Do you want to see the laws already available [n]?
enter replacement law:  plus(A,B,C), plus(C,D,E) -> plus(A,F,E), plus(B,D,F).

| ?- replace_goal(10,[3,7],1).

by replacing goal [plus(A,B,C),plus(C,D,E)] by [plus(A,F,E),plus(B,D,F)]
in clause 10 using law 1 we get:
11. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- s_p(B,D,I,K), mult(A,C,L),
          plus(L,M,J), plus(K,N,M), s_p(F,H,I,O), mult(E,G,P), plus(P,O,N).
```

```
| ?- add_law.
Do you want to see the laws already available [n]?
enter replacement law:  plus(A,B,C) -> plus(B,A,C).

| ?- replace_goal(11,[7],2).

by replacing goal [plus(A,B,C)] by [plus(B,A,C)] in clause 11 using law 2 we get:
12. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- s_p(B,D,I,K), mult(A,C,L),
         plus(L,M,J), plus(K,N,M), s_p(F,H,I,O), mult(E,G,P), plus(O,P,N).

| ?- add_law.
Do you want to see the laws already available [n]?
enter replacement law:  plus(A,F,E), plus(B,D,F) -> plus(A,B,C), plus(C,D,E).

| ?- replace_goal(12,[4,7],3).

by replacing goal [plus(A,B,C),plus(D,E,B)] by [plus(A,D,F),plus(F,E,C)]
in clause 12 using law 3 we get:
13. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- s_p(B,D,I,K), mult(A,C,L),
         plus(L,M,J), plus(K,N,O), plus(O,P,M), s_p(F,H,I,N), mult(E,G,P).

| ?- ts_fold(13, [1,6,4], 5).

by folding clause 13 w.r.t. atoms [1,6,4] using 5 we get:
14. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- p(B,D,F,H,I,K), mult(A,C,L),
         plus(L,M,J), plus(K,N,M), mult(E,G,N).
```

After the folding step, the current program consists of the two clauses 9 and 14 which avoid the two separate recursive calls used in the initial definition 5. Therefore, we end up the transformation sequence with the following final program:

```
9. p([ ],[ ],[ ],[ ],0,0).
14. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- p(B,D,F,H,I,K), mult(A,C,L),
         plus(L,M,J), plus(K,N,M), mult(E,G,N).
```

The transformation sequence shown in the above example fits with a usual derivation schema in the program transformation methodology based on the unfolding/folding rules. Indeed, a derivation usually begins with a definition introduction step followed by some unfolding steps, and it ends with some folding steps using the initial definition. In order to perform the final folding steps, we have used the associativity and commutativity of *plus* to rearrange the atoms in the body of clause 10. Those properties have been typed by the user, by applying the command `add_law`. They are called *replacement laws*, (or *laws* for short), and can be understood as lemmas used by the goal replacement rule (see rule 4 in Section 3.1). This rule is used to transform the body of a clause into a new body whose form is more convenient, in the sense that it allows subsequent transformation steps to be performed, like for instance, folding steps.
Since some laws might be useful on several occasions, the user can store them into separate files, referred to as 'theories', which can be loaded when needed. The set of laws available for applying the goal replacement rule during a derivation is called the *current theory*.

3 Transformations rules

The system deals with the class of *definite* programs, according to the standard terminology [6], that is, a program is a set of clauses of the form $H \leftarrow A_1, \ldots, A_n$, where $H, A_1, \ldots, A_n$ are atoms. However, a restricted form of negation is handled to allow the introduction of inequalities in the body of clauses, by application of the *case-split* rule. The head and the body of a clause C will be denoted $hd(C)$ and $bd(C)$. The set of variables occurring in a term t will be denoted $var(t)$.

3.1 Rules

Let $P_1, \ldots, P_k$ be a transformation sequence, and P_k be the current program. We should also consider some replacement laws. To transform P_k into a new program P_{k+1}, the following rules can be used:

1. Definition Introduction: Let C_i : $newp(\ldots) \leftarrow B_i$, with $i = 1, \ldots, n$ be a sequence of clauses such that the predicate *newp* does not occur in $P_0, \ldots, P_k$. By the definition introduction rule, we may get P_{k+1} by adding to P_k the sequence $C_1, \ldots, C_n$.

2. Unfolding: Let C : $H \leftarrow Left, A, Right$ be a clause in P_k, where *Left* and *Right* denote possibly empty sequences of atoms. Let $D_1, \ldots, D_n$ be all clauses in a program P_j, with $j \leq k$, such that A is unifiable with $hd(D_1)$, ..., $hd(D_n)$, with most general unifier $\theta_1, \ldots, \theta_n$ respectively. By unfolding C w.r.t. A using P_j, we derive the new n clauses C_i : $(H \leftarrow Left,\ bd(D_i),\ Right)\ \theta_i$, for $i = 1, \ldots, n$, and get the new program P_{k+1} by replacing C by $C_1, \ldots, C_n$ in P_k.

   There is no condition for the application of this rule, and it is possible to unfold a clause w.r.t. any atom of its body (unlike [2] which checks the conditions of deterministic unfolding and non-left propagating unfolding, in case a non-leftmost atom of a body is selected for unfolding). Notice that our definition assumes that there is at least one clause in P_j whose head unifies with the atom selected for unfolding. If this is not the case, that is, if the selected atom does not unify with the head of any clause of the initial program, we must specify whether:
   (*) no transformation step is performed and the current program is left unchanged, or
   (**) the clause which contains the selected atom is deleted from the current program.

   In our implementation, we provide two variants of the unfolding rule to let the user choose between the above two options. The second option can be seen as an extension of the first one which turns an unfolding step into a clause deletion step. Clause deletion can also be done separately by using rule 10 below.

3. Folding: Let $C_1, \ldots, C_n$ and $D_1, \ldots, D_n$ be two subsequences of clauses in P_k and P_j respectively, with $j \leq k$. Suppose that there exists a positive literal A such that, for $i = 1, \ldots, n$:

   (a) $hd(D_i)$ is unifiable with A via an m.g.u. θ_i,

   (b) C_i is the clause $(H \leftarrow Left, bd(D_i), Right)\ \theta_i$,

   (c) for any clause D of P_j not in $D_1, \ldots, D_n$, $hd(D)$ is not unifiable with A.

   By folding $C_1, \ldots, C_n$ using $D_1, \ldots, D_n$ in P_j, we derive the clause C : $H \leftarrow Left, A, Right$, and we get the new program P_{k+1} by replacing $C_1, \ldots, C_n$ by C in P_k.

4. Goal Replacement: Let C : $H \leftarrow F, S, G$ be a clause in P_k, where F, S and G are sequences of atoms. Let T be a sequence of atoms such that $var(H, F, G) \cap var(S) = var(H, F, G) \cap var(T)$. By replacement of S in C using the *law* $S \to T$, we derive the new clause R : $H \leftarrow F, T, G$ and we get P_{k+1} by replacing C by R in P_k.

5. Goal Rearrangement: Let C : $H \leftarrow A_1, \ldots, A_i, \ldots, A_n$ be a clause in P_k. We may derive a new clause M by moving literal A_i to position j, with $1 \leq j \leq n$, in the body of C, that is, M : $H \leftarrow A_1, \ldots, A_{j-1}, A_i, A_j, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n$ (assuming that $j \leq i$). We get P_{k+1} by replacing C by M in P_k.

6. Goal Deletion: Let C : $H \leftarrow F, A, G$ be a clause in P_k. We may derive a new clause M by deleting A in the body of C, thereby deriving M : $H \leftarrow F, G$, and get P_{k+1} by replacing C by M in P_k.

7. Goal Addition: Let C : $H \leftarrow A_1, \ldots, A_i, \ldots, A_n$ be a clause in P_k, and let G be a sequence of atoms. We may derive a new clause M by inserting G to position i in the body of C, thereby deriving M : $H \leftarrow A_1, \ldots, G, A_i, \ldots, A_n$, and get P_{k+1} by replacing C by M in P_k.

8. Definition Elimination: Let $C_i : q(\ldots) \leftarrow B_i$, with $i = 1, \ldots, n$ be a sequence of clauses such that the predicate q does not occur in P_0, and such that q does not occur in P_k in a clause different from $C_1, \ldots, C_n$. By the definition elimination rule, we may get P_{k+1} by deleting in P_k the sequence $C_1, \ldots, C_n$.
9. Clause Deletion: Let C be a clause in P_k. We may derive P_{k+1} by deleting C in P_k.
10. Deletion of clauses with finitely failed body: Let $C : H \leftarrow \mathit{Left}, A, \mathit{Right}$ be a clause in P_k, such that A unifies with no head of clause in P_k. We get the new program P_{k+1} by deleting C in P_k. This rule can be viewed as a particular case of rule 9.
11. Clause Addition: Let C be a clause. We may derive P_{k+1} by adding C to P_k.
12. Clause Replacement: Let $C_1, \ldots, C_n$ be a sequence of clauses in P_k. Let $D_1, \ldots, D_m$ be a sequence of clauses. We may derive P_{k+1} by replacing $C_1, \ldots, C_n$ by $D_1, \ldots, D_m$ in P_k.
13. Clause Rearrangement: Let C_1, C_2 be a sequence of clauses in P_k. We may derive P_{k+1} by replacing C_1, C_2 by C_2, C_1 in P_k.
14. Clause Subsumption: Let C and D be two clauses in P_k such that there exists a substitution θ and a sequence of atoms S such that $hd(C) = hd(D)\theta$, and $bd(C) = bd(D)\theta, S$. Then by clause subsumption, we get the new program P_{k+1} by deleting C in P_k.
15. Case Split: Let C: $H \leftarrow \mathit{Body}$ be a clause in P_k. Let X be a variable occurring in H and t be a term without occurrences of X. By case split of C w.r.t. $\{X/t\}$, we derive the following two clauses $C1 : (H \leftarrow \mathit{Body})\{X/t\}$ and $C2 : H \leftarrow X \neq t, \mathit{Body}$, and we get P_{k+1} by replacing C by $C1$ and $C2$ in P_k.

3.2 Semantics preservation

In order to preserve a given semantics during a tranformation sequence, some additional conditions for the applicability of the above rules are necessary. For instance, the goal replacement rule which allows us to replace a goal S by a goal T, is correct (for a suitable definition of correctness) if the two goals S and T are equivalent w.r.t. the semantics under consideration [8]. The clause deletion rule preserves the least Herbrand model of the program P_k if the deleted clause is subsumed by a different clause in P_k (see rule 14), or if it is a tautology, that is, a clause of the form $p(t) \leftarrow p(t)$. The goal rearrangement rule which allows us to move an atom within the body of a clause preserves the least Herbrand semantics of a program, but it might not preserve the Prolog semantics.

To prove the equivalence between two goals, and therefore to check the condition of applicability of the replacement rule, there exists a method based on unfold/fold transformation rules, but in general the applicability condition is an undecidable property [8]. Our transformer does not prevent the user from applying rules which do not preserve the least Herbrand semantics, and in general, the correctness of a derivation is left to the user's responsability. In particular, the correctness of the laws used by the goal replacement rule is assumed, and there are no conditions associated with the rules which allow for addition and deletion of goals and clauses. However, the partial correctness w.r.t. the least Herbrand semantics is ensured for every transformation sequence that uses only the rules 1, 2, 3, 5, 14, and 10, of Section 3.1, as it is shown in [8]. For a formal presentation of the correctness, w.r.t. various semantics, of the above transformation rules, apart from the case split rule, the reader may refer to [8]. The use and the issue of the correctness of the case split rule is discussed in [9, 12].

4 Representation of the transformation process

4.1 Transformation tree

The process of constructing a transformation sequence is nondeterministic, because many transformation rules may be applied to a given program. Thus, we represent this process as a tree, called the *transformation tree*. In a transformation tree, every node is labelled by a program and a term which indicates the transformation rule which has been applied to get the corresponding program.

Let P be a program in a transformation tree. The *history* of P is the sequence of programs and transformation rules associated with the branch from the initial program to P. The user may know the history of a program by issuing the command `show_history`. At the end of the derivation of the `scalar_product` example (Section 2), an application of this command gives the following result:

```
| ?- show_history.

P1: Progs/scalar_product
P2: by adding def 5 to P1
P3: by unfolding cl 5 in P2 wrt atom 1 using P1
P4: by unfolding cl 6 in P3 wrt atom 1 using P1
P5: by unfolding cl 8 in P4 wrt atom 1 using P1
P6: by unfolding cl 7 in P5 wrt atom 4 using P1
P7: by replacing atoms [3,7] in cl 10 in P6 using law 1
P8: by replacing atoms [7] in cl 11 in P7 using law 2
P9: by replacing atoms [4,7] in cl 12 in P8 using law 3
P10: by folding cl 13 in P9 w.r.t atoms [1,6,4] using 5
```

The corresponding transformation tree is depicted in Figure 1 (top).
Notice that the application of the command `add_law`, which is used three times in our example, does not appear in the history of the final program. Actually, since this command does not transform a program into another program, its application is not considered as a new step in the transformation sequence. This remark holds also for the command `show_history` illustrated above, and for many other commands (see Section 6.2),

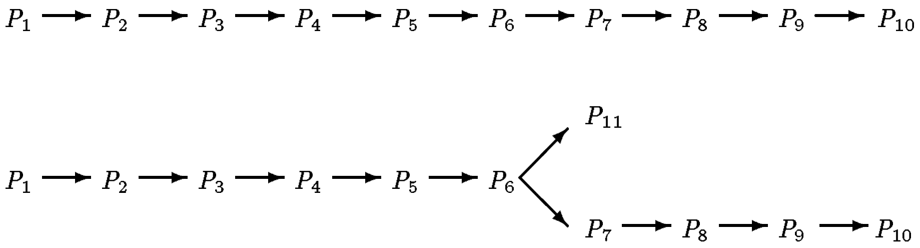

Figure 1: Transformation tree before (top) and after (bottom) backtracking on P_6

The representation of a transformation process by a tree allows the user to go back to any program in the transformation tree and to apply a new sequence of transformation rules. The new subderivation will enrich the current tree with a new branch, while the old one (if any) remains. For instance, at the end of the derivation of the `scalar_product` example, the user may want to go back to program P_6. To see the content of program P_6, he may first use the command `show_program` as follows:

```
| ?- show_program(6).

1. s_p([ ],[ ],0,0).
2. s_p([A|B],[C|D],s(E),F) :- s_p(B,D,E,G), mult(A,C,H), plus(H,G,F).
3. plus(0,A,A).
4. plus(s(A),B,s(C)) :- plus(A,B,C).
9. p([ ],[ ],[ ],[ ],0,0).
10. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- s_p(B,D,I,K), mult(A,C,L),
      plus(L,K,M), s_p(F,H,I,N), mult(E,G,O), plus(O,N,P), plus(M,P,J).
```

Then, to move to program P_6, and to unfold clause 10, he may issue the following commands:

```
| ?- set_current_program(6), unfold(10,1).

by unfolding clause 10 w.r.t. atom 1 we get:
15. p([A],[B],[C|D],[E|F],s(0),G) :- mult(A,B,H), plus(H,0,I),
          s_p(D,F,0,J), mult(C,E,K), plus(K,J,L), plus(I,L,G).
16. p([A,B|C],[D,E|F],[G|H],[I|J],s(s(K)),L) :- s_p(C,F,K,M),
          mult(B,E,N), plus(N,M,O), mult(A,D,P), plus(P,O,Q),
          s_p(H,J,s(K),R), mult(G,I,S), plus(S,R,T), plus(Q,T,L).
```

The command `set_current_program` illustrated above, allows the user to choose in a transformation tree any program as the starting point for a new subderivation. In particular he can backtrack to an ancestor of the current node, and this operation can be thought of as an undo action, except that the nodes corresponding to the undone steps are not removed from the tree. The resulting expanded tree is depicted in Figure 1 (bottom). The new program is program P_{11} whose history can be shown as follows:

```
| ?- show_history.

P1: Progs/scalar_product
P2: by adding def 5 to P1
P3: by unfolding cl 5 in P2 wrt atom 1 using P1
P4: by unfolding cl 6 in P3 wrt atom 1 using P1
P5: by unfolding cl 8 in P4 wrt atom 1 using P1
P6: by unfolding cl 7 in P5 wrt atom 4 using P1
P11: by unfolding cl 10 in P6 wrt atom 1 using P1
```

For conveniency, the user is provided with a command for saving the detailled history of a derivation in a file. It is a text file, which lists all programs and transformation rules of the corresponding transformation sequence, as above, but with the content of each program and possibly additional comments. It is also possible to save a transformation tree for restoring it at a later time, that is, to recover the tree without having to redo the corresponding transformation steps. These facilities and others will be described in Section 6.2.

4.2 Internal representation of data structures

As we have seen in the examples illustrating our transformer so far, user's commands are typed at the prompt '`| ?-`', that is, the application of a transformation rule corresponds to the execution of a goal at Prolog top-level.

From an implementation point of view, in order to keep track of the history of a derivation, the system uses dynamic relations whose definitions, a collection of facts, can be updated via the *assert* and *retract* Prolog built-in predicates. The collection of these dynamic relations describe the current state and the history of a derivation.
A transformation sequence may be initialized by one of the following two commands (se Section 6.2.1):

- `start` which allows the user to select an initial program,
- `load_session` which allows the user to restore a transformation tree previously saved by the command `save_session`.

When an initial program is selected, a tree made out of a root node, labelled by the initial program is created. When a session is restored, the transformation tree associated with the derivation of the corresponding session is recovered. Then, after each application of a transformation rule, the

current tree is expanded by creating a new node in the tree, a son of the node corresponding to the program on which the transformation rule has been applied.
The nodes of a tree are represented by atoms of the form `node(N,M,S,R,L)` where `N` is a node number (`1` for the root), `M` is the node number of the father of `N` (`0` for the root of the father), `S` is the list of the node numbers which are sons of `M`, `R` is a term identifying the rule applied on the program at node `M` to produce the program at node `N` (`0` for the root), and `L` is the list of the clause numbers in the program at node `N`. Clauses are represented by atoms of the form `clause3(N,C,I)`, where `C` is a term of the form `(Head,Body)` where `Head` and `Body` are an atom and a list of atoms, respectively. `N` is a clause number, that is, an integer. `I` may be i) the constant `prog` if `C` belongs to the initial program, or ii) the term `[def,C]` if `C` has been introduced by a definition introduction step using the command `ts_define` (see Section 6.1.1), or iii) the term `[gen_def|Cs]` if `C` has been introduced by a definition introduction step involving clauses `Cs` using command `gen_define` (see Section 6.1.1), or iv) the constant `nil` otherwise. In our introductory example, at the execution of the `start` command, the following facts are asserted:

```
node(1,0,[],0,[1,2,3,4]).
clause3(1,(s_p([ ],[ ],0,0),[ ]),prog).
clause3(2,(s_p([A|B],[C|D],s(E),F),[s_p(B,D,E,G),mult(A,C,H),plus(H,G,F)]),prog).
clause3(3,(plus(0,A,A),[ ]),prog).
clause3(4,(plus(s(A),B,s(C)),[plus(A,B,C)]),prog).
```

At the end of the derivation, the relation `node/5` consists of the following facts:

```
node(10,9,[ ],(13,(fold,[1,6,4],5),[14]),[1,2,3,4,9,14]).
node(9,8,[10],(12,(rep,[4,7],3),[13]),[1,2,3,4,9,13]).
node(8,7,[9],(11,(rep,[7],2),[12]),[1,2,3,4,9,12]).
node(7,6,[8],(10,(rep,[3,7],1),[11]),[1,2,3,4,9,11]).
node(6,5,[7],(7,(unf,4),[10]),[1,2,3,4,9,10]).
node(5,4,[6],(8,(unf,1),[9]),[1,2,3,4,9,7]).
node(4,3,[5],(6,(unf,1),[8]),[1,2,3,4,8,7]).
node(3,2,[4],(5,(unf,1),[6,7]),[1,2,3,4,6,7]).
node(2,1,[3],(def,[5]),[1,2,3,4,5]).
node(1,0,[2],0,[1,2,3,4]).
```

The current theory, which is empty until the first application of the command `add_law`, is represented, at the end of the derivation, by the following facts:

```
law(1,([plus(A,B,C),plus(C,D,E)],[plus(A,F,E),plus(B,D,F)])).
law(2,([plus(A,B,C)],[plus(B,A,C)])).
law(3,([plus(A,B,C),plus(D,E,B)],[plus(A,D,F),plus(F,E,C)])).
```

Since we never remove nodes from a transformation tree, a clause associated with the relation `node/5` is never retracted. On the contrary, a clause associated with the relation `law/2` may be retracted, by applying the command `delete_law_from_theory` (see Section 6.2.2). To allow for the possibility of retracting a law, is of little significance for the moment, since the user has to specify which law to use when applying the goal replacement rule, and therefore he may always ignore the laws he does not intend to use. However, we may think of enhancing our system with a strategy which would automatically apply goal replacements, according to a possibly variable set of laws. In this case it would be convenient to control, during the derivation, the availability of laws via some assert and retract operations. Currently, the only interest the user may have to retract a law from the current theory is in the case he wants to save it in a file by applying the command `save_current_theory` (see Section 6.2.4).
Apart from the relations `clause3/3`, `node/5`, and `law/2`, there exist some other relations related to data-structures which are updated, after each transformation step, via some assert and retract

operations. For instance, `maxnode/1`, `max_clause/1`, `max_law/1`, are used for the numbering of programs, clauses, and laws respectively, and `current_prog/2` is used for keeping track of the current program, defined by a number and a list of clause numbers.

5 Running the transformer

5.1 Installation

To install the SICStus (resp. Quintus) version of the transformer one needs the package `t6.tar.gz` (resp. `t5.tar.gz`) to be uncompressed and unpacked as usual. For instance, for the SICStus package, one must type the following commands:

```
> gunzip t6.tar.gz
> tar xvf t6.tar
```

This will create a directory `TRANSFORMER/` containing other directories and files as described below.

- the directory `Progs/` contains the user programs intended to be transformed, that is, those programs which can be selected to begin a transformation sequence. The menu proposed to the user when he issues the command `start` (see Section 6.2.1), with no argument, is made out of all files present in `Progs/`. This means that the user must preliminarily put in this directory the program which he intends to transform. Programs must be written in the standard Prolog syntax. The files `scalar_product`, `rotate_leftdepth`, `queens`, `match` and `multi_match` are given to allow the user to replay easily the examples given in this report.
- the directory `Final_Progs/` contains programs saved during a transformation process via the command `save_prog` (see Section 6.2.4).
- the directory `Theories/` contains theories, that is, sets of laws which are intended to be used for the replacement rule. In order to make a law of a theory available for the replacement rule, the associated theory must be first selected. This can be done via the command `select_law_in_theory` (see Section 6.2.2). Laws are rewrite rules according to the following syntax: $A_1, \ldots, A_n \rightarrow B_0, \ldots, B_m.$, where $A_1, \ldots, A_n, B_0, \ldots, B_m$ are atoms.
- the directory `Histories/` contains files associated with the history of a derivation that have been saved via the command `save_history` (see Section 6.2.4). These files provide a summary of a transformation sequence by listing all programs from the initial to the final one, with a comment saying how a program has been obtained from its predecessor. These files are for documentation only and are not supposed to be manipulated further by the system itself. Therefore the user is free to modify them in any way he likes.
- the directory `Sessions/` contains files associated with a state of a transformation that have been saved via the command `save_session` (see Section 6.2.4). These files are intended to be restored via the command `load_session` (see Section 6.2.1), and are not supposed to be modified by the user.
- the directory `src/` contains the source files of the transformer (see Appendix C).
- The file `determ_strat_parameters.pl` is a file containing the values of some parameters used by the strategy `determ`. It is consulted at each application of the strategy, and therefore, it must be modified according to the program to be transformed and the chosen parameters, before each application of the strategy. In Appendix C.17, we give two versions of this file which are used in the two examples of Section 7.3.1 and Section 7.3.3. The first version of the file is commented to allow the user to modify easily the parameters for other examples.

The transformer has been implemented in Quintus Prolog (Version 3.1.3) using several predicates of the Quintus libraries. Recently the code has been adaptated to SICStus Prolog (Version 3 # 5), so that the transformer is now available for both Quintus and SICStus Prolog.

5.2 Invoking the transformer

To use the transformer, the user must be in the directory `TRANSFORMER/`. The entry `t6` (resp. `t5`) typed at the Unix prompt runs a SICStus (resp. Quintus) Prolog application which outputs a menu and prompts the user as follow:

```
adp10{~/Transformer}:t6
     st.           < start a new transformation process >
     r.            < show available rules >
     s.            < show available strategies >
     c.            < show other available commands >
     helps.        < show this five lines menu >
| ?-
```

`st`, `r`, `s`, `c`, and `helps` are some commands that can be issued by the user at the prompt '`| ?- `', that is at the Prolog top-level. The other commands provided by the system are divided into the three categories 'rules', 'strategies' and 'other commands'. They can be shown by applying `r`, `s` and `c` respectively. Some of these commands have been illustrated in the introductory example. In Section 6, we describe the whole collection of transformation rules and other commands, and in Section 7 we present the three commands `uve`, `lap`, and `determ`, which correspond to strategies.

6 Top-level commands

All commands available to the user are atomic goals with arguments, if necessary. Arguments are generally integers or lists of integers, denoting clauses, atoms, programs, or laws. As we have seen in Section 4.2, clauses, programs and laws are automatically numbered. In a clause body, atoms are numbered according to their position, in increasing order, from left to right, starting from 1.

6.1 Transformation rules

In this section we present the commands of the transformer which allow for the application of the transformation rules presented in Section 3. They form the basic transformation system.

6.1.1 Definition introduction rules

- `ts_define`

  arguments: none.

  use: allows the user to enter a new definition, according to Tamaki and Sato's definition introduction rule, that is, the new definition must contain one clause only and use a new predicate in its head. If these conditions are not satisfied, the rule fails and the current program remains unchanged. This rule has no argument, and the user is prompted to enter the new definition, which is then appropriately numbered and echoed to him.

  shorthand: `ts_d`
  `ts_d` is a short for `ts_define`.

- `gen_define`

  arguments: none.

  use: generalizes `ts_define` by allowing the user to enter a new definition possibly made out of several clauses. The new clauses must use the same new predicate in their heads, otherwise the rule fails and the current program remains unchanged.

  shorthand: `d`
  `d` is a short for `gen_define`.

When possible, user's interactions are eased by associating default values to the required entries. They are indicated within square brackets at the end of messages, like `[n]` in the example below. If the user agrees with the default value, he may simply hit the RETURN key.

Example: We illustrate the use of the above two definition introduction rules, starting from an initial program named `ex1` made out of the two clauses $\{p(a) \leftarrow,\ \ p(b) \leftarrow q\}$. As in the example of Section 2, we have underlined the parts of the session which are typed by the user.

```
| ?- start(ex1).

Initial program:
1. p(a).
2. p(b) :- q.

| ?- ts_d.
enter definition clause:
|:  r :- q.

3. r :- q.

| ?- d.
Do you want to check the existing definitions [n]?  y
3.  r :- q.
enter clauses (use the same new predicate in the heads):
first clause:  s :- p(a).
Another definition [n]?  y
clause:  s :- r, p(b), p(c).
Another definition [n]?  y
clause:  s :- r, p(b), p(a).
Another definition [n]?

4. s :- p(a).
5. s :- r, p(b), p(c).
6. s :- r, p(b), p(a).
| ?-
```

6.1.2 Unfolding rules

- `unfold(C,A)`

  arguments:

  `C`: clause number in the current program.
  `A`: atom number in the body of `C`.

  use: unfolds clause `C` w.r.t. atom `A` using initial program `P1`. If no head of clause in `P1` unifies with atom A, clause deletion is not performed, and the transformation tree remains unchanged.

  shorthands: `u(C,A)`, `u(C)`, `u`

  `u(C,A)` is a short for `unfold(C,A)`.
  `u(C)` is a short for `u(C,1)`.
  `u` is the same as `u(C,A)`, except that the user is prompted to enter `C` and `A`.

- `unfold_d_i(Pi,C,A)`

  arguments:

  `Pi`: program number in the history of the current program.
  `C`: clause number in the current program.
  `A`: atom number in the body of `C`.

use: unfolds clause `C` w.r.t. atom `A` using program `Pi`. If atom `A` uses a predicate which is defined in program `Pi`, but is such that no head of clause unifies with atom `A`, then clause deletion is applied. Actually, `C` has a finitely failed body.

shorthands:

`ud_i(Pi,C,A), ud_i(Pi,C), ud_i, ud(C,A), ud(C), ud, ud_p(C,A), ud_p(C), ud_p`

`ud_i(Pi,C,A)` is a short for `unfold_d_i(Pi,C,A)`.

`ud_i(Pi,C)` is a short for `ud_i(Pi,C,1)`.

`ud_i` is the same as `ud_i(C,A)`, except that the user is prompted to enter `C` and `A`.

`ud(C,A)` is a short for `ud_i(1,C,1)`, that is, `Pi` is the initial program.

`ud(C)` is a short for `ud(C,1)`.

`ud` is the same as `ud(C,A)`, except that the user is prompted to enter `C` and `A`.

`ud_p(C,A)` is a short for `ud_i(Pn,C,1)`, where `Pn` is the current program.

`ud_p(C)` is a short for `ud_p(C,1)`.

`ud_p` is the same as `ud_p(C,A)`, except that except that the user is prompted to enter `C` and `A`.

- `unfold_d_facts(C)`

  argument: `C`: clause number in the current program.

  use: unfolds clause `C` w.r.t atoms which are instances of unit clauses in the current program.

  shorthands: `ud_facts(C), ud_facts`

  `ud_facts(C)` is a short for `unfold_d_facts(C)`.

  `ud_facts` applies `ud_facts(C)` to every clause `C` in the current program.

- `unfold_d_eqs(C)`

  argument: `C`: clause number in the current program.

  use: unfolds clause `C` w.r.t. all equalities (if any) of its body.

  shorthands: `ud_eqs(C), ud_eqs`

  `ud_eqs(C)` is a short for `unfold_d_eqs(C)`.

  `ud_eqs` applies `ud_eqs(C)`, to every clause `C` in the current program.

Example: The derivation from program `ex1` in Section 6.1.1 could be continued with the following unfolding steps:

```
| ?- u(4), u(5,2), u(6,2).

by unfolding clause 4 w.r.t. atom 1 we get:
7. s.
by unfolding clause 5 w.r.t. atom 2 we get:
8. s :- r, q, p(c).
by unfolding clause 6 w.r.t. atom 2 we get:
9. s :- r, q, p(a).

| ?- ud(8,3), ud(9,3).

deletion of clause 8 because of failure in body
by unfolding clause 9 w.r.t. atom 3 we get:
10. s :- r, q.

| ?- ud_i(2,10).

by unfolding clause 10 w.r.t. atom 1 using program 2 we get:
11. s :- q, q.
| ?-
```

Notice that the unfolding of clause 10 in the above derivation would not have been possible by using the command `ud(10)` since predicate `r` has been introduced by a definition rule, and therefore it is not defined in the initial program. It is defined in program `P2`, since `r` has been introduced at the first step of the derivation, by clause `3`. Notice finally that since the definition of predicate `r` has not been changed since then (in particular clause 3 has not been unfolded), it would have been equivalent to apply `ud_p(10)` instead of `ud_i(2,10)`.

6.1.3 Folding rules

- `ts_fold(C,As,D)`

  arguments:

  `C`: clause number in the current program.

  `As`: list of atom numbers in the body of `C`.

  `D`: clause number in the current program or in one of its ancestors, which must correspond to a clause introduced by definition (see Section 6.1.1).

  use: folds clause `C` using clause `D`, w.r.t. the atoms in list `As`. This folding rule is the one of Tamaki and Sato [13]. `As` must be ordered, so that, the atoms in `As` form, in the given order, an instance of the body of `D`.

  shorthand: `ts_f(C,As,D)`

  `ts_f(C,As,D)` is a short for `ts_fold(C,As,D)`.

- `fold(Cs1,Cs2)`

  arguments:

  `Cs1`: list of clause numbers in the current program.

  `Cs2`: list of clause numbers in the current program, or in one of its ancestors.

  use: folds the clauses in list `Cs1` using those in list `Cs2`. `Cs1` and `Cs2` must have same length, and must be ordered so that the i-th clause in `Cs1` is folded using the i-th clause in `Cs2`. No permutation in `Cs1` is attempted in case the given order does not allow the folding step. No permutation in the bodies of the clauses is attempted neither, that is, in order to fold clauses $C_1, \ldots, C_n$ using clauses $D_1, \ldots, D_n$, $bd(C_i)$ must be of the form $(\mathit{Left}, bd(D_i), \mathit{Right})\theta_i$, for each i, $1 \leq i \leq n$. This folding rule correspond to the one presented in Section 3 (rule 3).

  shorthand: `f(Cs1,Cs2)`

  `f(Cs1,Cs2)` is a short for `fold(Cs1,Cs2)`.

- `rearrange_and_fold(Cs1,Cs2)`

  arguments:

  `Cs1`: list of clause numbers in the current program.

  `Cs2`: list of clause numbers in the current program, or in one of its ancestors.

  use: extends `fold(Cs1,Cs2)` in the following sense. First, it does not require that `Cs1` and `Cs2` are ordered, that is, if `fold(Cs1,Cs2)` fails, it will also try `fold(PCs1,Cs2)` where `PCs1` is a permutation of `Cs1`. Second, it allows for permutations in the bodies of the clauses, that is, for each i, $1 \leq i \leq n$, it only requires that there exists a permutation of the atoms of $bd(C_i)$ which is of the form $(\mathit{Left}, bd(D_i), \mathit{Right})\theta_i$, for some substitution θ_i (cf. rule 3 in Section 3.1).

  shorthand: `r_f(Cs1,Cs2)`

  `r_f(Cs1,Cs2)` is a short for `rearrange_and_fold(Cs1,Cs2)`.

Example:
Below we illustrate the difference between the rules `fold/2` and `rearrange_and_fold(Cs1,Cs2)`, starting from an initial program named `ex2`. The first attempt of folding using rule `fold/2` fails because the bodies of clauses 3 and 4 are not of the form required by the definition of the folding

rule of Section 3. The folding step requires a prior reorganization of the bodies of clauses 3 and 4, which is performed by the rule `rearrange_and_fold/2`. Then we illustrate the use of the rule `ts_fold`, which requires a preliminary definition introduction step.

```
| ?- start(ex2).

Initial program:
1. p([a|A]) :- t(A), u(A), v(A).
2. p([b|A]) :- s(A), t(A), u(A).
3. p2(a,A) :- v(A).
4. p2(b,A) :- s(A).

| ?- f([1,2], [3,4]).

   Impossible folding.

| ?- r_f([1,2], [3,4]).

by folding clauses [1,2] using clauses [3,4] we get:
5. p([A|B]) :- p2(A,B), t(B), u(B).

| ?- ts_d.
enter definition clause:
|:  q(A,B) :- p2(A,B), u(B).

6. q(A,B) :- p2(A,B), u(B).

| ?- ts_f(5,[1,3],6).

by folding clause 5 w.r.t. atoms [1,3] using 6 we get:
7. p([A|B]) :- q(A,B), t(B).
```

6.1.4 Goal replacement rules

- `replace_goal(C,As,L)`

  arguments:
  - `C`: clause number in the current program.
  - `As`: list of atom numbers in the body of `C`.
  - `L`: law number in the current theory.

  use: replaces the goal made out of atoms `As` in the body of clause `C` by another goal according to the replacement law `L`. `As` must be ordered, so that, the atoms in `As` form, in the given order, an instance of the left hand side of `L`. It is thus replaced by the corresponding instance of the right hand side of `L`. The conditions on variables given in Section 3.1 (see rule 4) are checked.

  shorthand: `rep(C,As,L)`
  `rep(C,As,L)` is a short for `replace_goal(C,As,L)`.

- `move_lit(C,Pos1,Pos2)`

  arguments:
  - `C`: clause number in the current program.
  - `Pos1`: atom number in the body of `C`.
  - `Pos2`: atom number in the body of `C`.

  use: moves, in the body of clause `C`, the atom at position `Pos1` to position `Pos2`.

  shorthand: `mv(C,Pos1,Pos2)`
  `mv(C,Pos1,Pos2)` is a short for `move_lit(C,Pos1,Pos2)`.

- `add_goal(C)`

  argument: `C`: clause number in the current program.

  use: adds a goal in the body of clause `C`. The user is prompted to type the new goal and the position in the body of clause `C` where it must be added.

  shorthand: `add_g(C)`
  `add_g(C)` is a short for `add_goal(C)` .

- `delete_goal(C,As)`

  arguments:
  `C`: clause number in the current program.
  `As`: list of atom numbers in the body of `C`.

  use: deletes the goal made out of atoms `As` in the body of clause `C`.

  shorthand: `del_g(C,As)`
  `del_g(C,As)` is a short for `delete_goal(C,As)`.

6.1.5 Clause replacement rules

- `restriction(F/A)`

  arguments:
  `F`: predicate name occurring in the current program.
  `A`: arity of `F`.

  use: deletes from current program clauses that are not used in the definition of `F/A`.

  shorthand: `restr(F/A)`
  `restr(F/A)` is a short for `restriction(F/A)`.

- `delete_clauses(Cs)`

  argument: `Cs`: clause number, or list of clause numbers in the current program.

  use: deletes clause(s) `Cs`.

  shorthand: `del_cl(Cs)`
  `del_cl(Cs)` is a short for `delete_clauses(Cs)`.

- `delete_failing_body_clauses(Cs)`

  argument: `Cs`: clause number, or list of clause numbers in the current program.

  use: deletes among clause(s) `Cs` those which have a failing body, that is, which contain an atom whose predicate is defined in the current program but which does not unify with the head of any clause.

  shorthands: `del_f_cl(Cs), del_f_cl`
  `del_f_cl(Cs)` is a short for `delete_failing_body_clauses(Cs)`.
  `del_f_cl` applies `del_f_cl(C)` to every clause `C` in the current program.

- `add_clause`

  arguments: none.

  use: adds a clause to the current program. The user is prompted to type the new clause and the position in the current program where it must be added.

  shorthand: `add_cl`
  `add_cl` is a short for `add_clause`.

- `replace_clause(C)`

  argument: `C`: clause number in the current program.

  use: replaces clause `C` by another clause. The user is prompted to type the replacing clause.

  shorthand: `rep_cl(C)`
  `rep_cl` is a short for `replace_clause`.

- `move_clause(C)`

  argument: `C`: clause number in the current program.

  use: moves clause `C` within the current program. The user is prompted to give the new position.

  shorthand: `mv_cl(C)`
  `mv_cl` is a short for `move_clause`.

- `subsumption(Cs)`

  argument: `Cs`: clause number, or list of clause numbers in the current program.

  use: applies subsumption to clause(s) `Cs`, that is, deletes among `Cs` all clauses that are subsumed by another clause in the current program.

  shorthands: `subs(Cs), subs`
  `subs(Cs)` is a short for `subsumption(Cs)`.
  `subs` is a short for `subs(Cs)`, where `Cs` are all clauses in the current program.

- `split_clause(C,A)`

  arguments:
  `C`: clause number in the current program.
  `A`: term which must be an instance of the head of clause `C`.

  use: splits clause `C` w.r.t. atom `A` in the following way: If `C` is the clause with head `H` and body `Body`, and if `A` is an instance of `H` via the substitution $\{X_1/t_1, \ldots, X_n/t_n\}$, clause `C` is replaced by the following two clauses:
  $C1: \ (H \leftarrow Body)\{X_1/t_1, \ldots, X_n/t_n\}$, and
  $C2: \ H \leftarrow X_1 \neq t_1, \ldots, X_n \neq t_n, Body$,
  otherwise it fails.

  shorthand: `split(C,A)`
  `split(C,A)` is a short for `split_clause(C,A)`.

Example: We now give an example of a derivation which uses some rules described in Sections 6.1.4 and 6.1.5. We start from an initial program named `ex3`.

```
| ?- start(ex3).

Initial program:
1. p(A) :- q(A,B), r(B).
2. q(a,b).
3. r(b).
4. s(a).

| ?- del_cl(4).
deletion of clause 4
| ?- split(1, p(a)).

by case splitting clause 1 we get:
5. p(a) :- q(a,A), r(A).
6. p(A) :- \+A=a, q(A,B), r(B).

| ?- add_cl.
enter new clause:  p(a) :- q(a,A).
at which position ?  [at last position]:  2

Addition of clause 7:
5. p(a) :- q(a,A), r(A).
7. p(a) :- q(a,A).
```

```
6. p(A) :- \+A=a, q(A,B), r(B).
2. q(a,b).
3. r(b).

| ?- mv(6,1,3).

by moving atom 1 to position 3 in clause 6 we get:
8. p(A) :- q(A,B), r(B), \+A=a.

| ?- split(8, p(b)).

by case splitting clause 8 we get:
9. p(b) :- q(b,A), r(A), \+b=a.
10. p(A) :- \+A=b, q(A,B), r(B), \+A=a.

| ?- del_g(10,[1,4]).

by deleting goal [1,4] in clause 10 we get:
11. p(A) :- q(A,B), r(B).

| ?- add_g(11).
Goal:  s(a,b).
position of new goal [3]:  2
by adding goal s(a,b) in clause 11 at position 2 we get:
12.  p(A) :- q(A,B), s(a,b), r(B).
| ?- subs.
clause 5 subsumed by 7
| ?- del_f_cl
deletion of clause 9 because of failure in body
Warning:  undefined predicate s/2
| ?-
```

6.2 Other commands

In this section, we present some commands which are not directly related to the application of a transformation rule, like for instance, the command `start` which allows the user to select an intial program, or the command `show_history` which shows the history of a derivation. Most of these commands aim to give or save information.

6.2.1 Commands for starting or navigating the transformation tree

- `start(F)`

  argument: `F`: name of a file in directory `Progs/`.

  use: starts a transformation with `F` as initial program.

  shorthands: `st(F), st`

  `st(F)` is a short for `start(F)`.

  `st` is similar to `st(F)`, except that `F` is selected by the user via a menu which lists all files present in directory `Progs/`.

- `load_session`

  arguments: none.

  use: restores a transformation tree which is selected by the user via a menu which lists all files present in directory `Sessions/`. This directory consists of the files automatically produced by the command `save_session` and should not be modified by the user.

  shorthand: `load_s`

  `load_s` is a short for `load_session`.

- `set_current_program(P)`

  argument: `P`: program number in the transformation tree.

  use: moves to program `P`, that is program `P` becomes the new current program.

  shorthand: `set(P)`
  `set(P)` is a short for `set_current_program(P)`.

6.2.2 Commands for selecting or editing replacement laws

- `add_law_in_theory(T)`

  argument: `T`: name of a file in directory `Theories/`.

  use: adds a law to theory `T`. The user is prompted to enter the new law. It is added at the end of file `T`, and `T` is saved with the corresponding change.

  shorthands: `add_l(T), add_l`
  `add_l(T)` is a short for `add_law_in_theory(T)`.
  `add_l` adds a law to the current theory.

- `delete_law_from_theory(T)`

  argument: `T`: name of a file in directory `Theories/`.

  use: deletes a law from theory `T`. The law is selected by the user via a menu which lists all laws of theory `T`. It is removed from file `T`, and `T` is saved with the corresponding changes.

  shorthands: `del_l(T), del_l`
  `del_l(T)` is a short for `delete_law_from_theory(T)`.
  `del_l` deletes a law from the current theory.

- `select_law_in_theory`

  arguments: none.

  use: adds a law from a theory `T` to current theory. `T` is selected by the user via a menu which lists all files of directory `Theories/`, and the law to be added is selected via a menu which lists all laws in theory `T`.

  shorthand: `sel_l`
  `sel_l` is a short for `select_law_in_theory`.

6.2.3 Commands for showing information

- `show_program(Ps)`

  argument: `Ps`: program number, or list of program numbers in the transformation tree.

  use: shows program(s) `Ps`.

  shorthands: `p(Ps), p1, p`
  `p(Ps)` is a short for `show_program(Ps)`.
  `p1` shows the initial program.
  `p` shows the current program.

- `show_short_program(Ps)`

  argument: `Ps`: program number, or list of program numbers in the transformation tree.

  use: `show_short_program(Ps)` is similar to `show_program(Ps)`, except that only clauses that are not present in the initial program are shown.

  shorthands: `pp(Ps), pp`
  `pp(Ps)` is a short for `show_short_program(Ps)`.
  `pp` is a short for `pp(Pn)` where `Pn` is the current program.

- `show_history(P)`

  argument: `P`: program number in the transformation tree.

  use: shows the history of program `P`.

  shorthands: `h(P), h`
  `h(P)` is a short for `show_history(P)`.
  `h` is a short for `h(Pn)`, where `Pn` is the current program.

- `show_clauses(Cs)`

  argument: `Cs`: clause number, or list of clause numbers in the current program.

  use: shows clause(s) `Cs`.

  shorthand: `cls(Cs)`
  `cls(Cs)` is a short for `show_clauses(Cs)`.

- `show_definitions`

  arguments: none.

  use: shows clauses that have been introduced so far by definition introduction, that is, by applying the commands `ts_d` or `d`.

  shorthand: `defs`
  `defs` is a short for `show_definitions`.

- `show_laws`

  arguments: none.

  use: shows current theory.

  shorthand: `l`
  `l` is a short for `show_laws`.

- `directory_progs`

  arguments: none.

  use: shows files present in the directory `Progs/` (via the unix command `ls`).

  shorthand: `dir_p`
  `dir_p` is a short for `directory_progs`.

- `directory_theories`

  arguments: none.

  use: shows files present in the directory `Theories/` (via the unix command `ls`).

  shorthand: `dir_t`
  `dir_t` is a short for `directory_theories`.

- `directory_sessions`

  arguments: none.

  use: shows files present in the directory `Sessions/` (via the unix command `ls`).

  shorthand: `dir_s`
  `dir_s` is a short for `directory_sessions`.

- `directory_final_progs`

  arguments: none.

  use: shows files present in the directory `Final_Progs/` (via the unix command `ls`).

  shorthand: `dir_f`
  `dir_f` is a short for `directory_final_progs`.

6.2.4 Commands for saving information

- `save_prog(P)`

  argument: `P`: program number in the transformation tree.

  use: saves program `P` in directory `Final_Progs/`.

  shorthands: `s_p(P)`, `s_p`

  `s_p(P)` is a short for `save_prog(P)`.
  `s_p` is a short for `s_p(Pn)`, where `Pn` is the current program.

- `save_history(P)`

  argument: `P`: program number in the transformation tree.

  use: saves the history of program `P` in directory `Histories/`.

  shorthands: `s_h(P)`, `s_h`

  `s_h(P)` is a short for `save_history(P)`.
  `s_h` is a short for `s_h(Pn)`, where `Pn` is the current program.

- `save_current_theory`

  arguments: none.

  use: saves the current theory in directory `Theories/`.

  shorthand: `s_th`

  `s_th` is a short for `save_current_theory`.

- `save_session`

  arguments: none.

  use: saves the current derivation in directory `Sessions/`.

  shorthand: `s_sess`

  `s_sess` is a short for `save_session`.

6.2.5 Commands for adding comments to the history of a derivation

- `insert_comment_history`

  arguments: none.

  use: inserts user's comments to the history of the current program. Running this command prompts the user to type a sequence of characters ended by a full stop. The corresponding text will be added to the current history, that is, it will appear in the output file produced by applying the command `save_history`.

  shorthand: `com_h`

  `com_h` is a short for `insert_comment_history`.

6.2.6 Commands for getting help

- `helps`

  arguments: none.

  use: shows a general help menu.

- `rules`

  arguments: none.

  use: shows a menu with all rules described in Section 6.1.

  shorthand: `r`

  `r` is a short for `rules`.

- `commands`

  arguments: none.

  use: shows a menu with all commands described in Section 6.2.

  shorthand: `c`

  `c` is a short for `commands`.

- `strategies`

  arguments: none.

  use: shows a menu with all strategies described in Section 7.

  shorthand: `s`

  `s` is a short for `strategies`.

- `man(X)`

  argument: `X`: name of a rule, a command, or a strategy.

  use: shows details about the use of rule, command, or strategy `X`.

Example: Below, we illustrate some of the above commands with reference to the derivation shown in Section 2. We recall that, in that example, three laws have been added using the command `add_l`, and therefore the current theory at the end of the derivation is made out of these three laws. We show how these laws can be saved in a theory, and how some addition and deletion of laws can be further performed. We assume the existence of a theory, called `list_theory`, already present in directory `Theories/`.

```
| ?- l.

1. plus(A,B,C), plus(C,D,E) -> plus(A,F,E), plus(B,D,F).
2. plus(A,B,C) -> plus(B,A,C).
3. plus(A,B,C), plus(D,E,B) -> plus(A,D,F), plus(F,E,C).

| ?- dir_t.
1) list_theory
Do you want to look at a theory [n]?

| ?- s_th.
Name of the new theory :  plus_theory.
Current theory saved in Theories/plus_theory

| ?- dir_t.

1) plus_theory
2) list_theory
Do you want to look at a theory  [n]?

| ?- add_l(plus_theory).
Do you want to see the already existing laws [n]?  y

plus(A,B,C), plus(C,D,E) -> plus(A,F,E), plus(B,D,F).
plus(A,B,C) -> plus(B,A,C).
plus(A,B,C), plus(D,E,B) -> plus(A,D,F), plus(F,E,C).

enter new law:  plus(A,B,0) -> A=0, B=0.
Another law to be added [n]?
New theory saved in file:  Theories/plus_theory
```

```
| ?- del_l(plus_theory).

1. plus(A,B,C), plus(C,D,E) -> plus(A,F,E), plus(B,D,F).
2. plus(A,B,C) -> plus(B,A,C).
3. plus(A,B,C), plus(D,E,B) -> plus(A,D,F), plus(F,E,C).
4. plus(A,B,0) -> A=0, B=0.

enter law to delete [none]:  2
plus_theory has been updated.
The previous version has been saved in file old_plus_theory
Another law to be deleted [n]?

| ?- dir_t.

1) old_plus_theory
2) plus_theory
3) list_theory

Do you want to look at a theory [n]?  y
Which one ?  [none]:2

plus(A,B,C), plus(C,D,E) -> plus(A,F,E), plus(B,D,F).
plus(A,B,C), plus(D,E,B) -> plus(A,D,F), plus(F,E,C).
plus(A,B,0) -> A=0,B=0.

| ?- sel_l.

1) old_plus_theory
2) plus_theory
3) list_theory

Theory number [no selection]:  2

1. plus(A,B,C),plus(C,D,E) -> plus(A,F,E),plus(B,D,E)
2. plus(A,B,C),plus(D,E,B) -> plus(A,D,F),plus(F,E,C)
3. plus(A,B,0) -> A=0,B=0

Law number [no selection]:  3
Another law to select [n]?

| ?- l.

1. plus(A,B,C), plus(C,D,E) -> plus(A,F,E), plus(B,D,F).
2. plus(A,B,C) -> plus(B,A,C).
3. plus(A,B,C), plus(D,E,B) -> plus(A,D,F), plus(F,E,C).
4. plus(A,B,0) -> A=0, B=0.
```

Finally, to get help about a command, one may use the command **man**, with the name of the command to be described as argument. For example, to get some help about the command **subs**, we should do as follows:

```
| ?- man(subs).

subs(Cs):
     Applies subsumption to clauses Cs, that is, deletes among Cs
     all clauses that are subsumed by another clause in Cs.

subs:
     is a short for subs(Cs), that is, applies subsumption to current program.
```

```
Example:
     Assume that the current program is made out of the following clauses
     5. p(a) :- q(a,A), r(A).
     6. p(A) :- +A=a, q(A,B), r(B).
     7. p(a) :- q(a,A).
     | ?- subs.
     clause 5 subsumed by 7
     | ?- p.
     6. p(A) :- +A=a, q(A,B), r(B).
     7. p(a) :- q(a,A).
     | ?-
```

7 Transformation strategies

In this section we present three strategies, namely `uve`, `lap`, and `determ`, that can be applied automatically by the system.

7.1 A strategy for removing unnecessary variables

The strategy `uve` implements the unnecessary variables elimination strategy described in [11]. An unnecessary variable of a clause C is either a variable which occurs in the body of C and not in its head, or it occurs more than once in the body of C. The presence of unnecessary variables may cause inefficiency by producing redundant computations. Therefore, we may expect to increase efficiency of a program which contains unnecessary variables by applying this strategy. We recall however that the elimination of unnecessary variables is not always possible and that the problem of deciding whether or not the Elimination Procedure terminates is not solvable [11]. The Elimination Procedure is based on a repeated application of some unfolding, definition, and folding steps to each clause with unnecessary variables. In order to execute the procedure fully automatically, we need to specify how to unfold, that is, to define a *selection* function which decides the atoms w.r.t. which clauses are unfolded.

In our implementation, the selection of atoms w.r.t. which clauses are unfolded is done interactively, that is, the execution of the `uve` strategy is performed semi-automatically. To run the strategy, the user must first load an initial program via the command `start` and then apply the command `uve`. Then, for each clause with unnecessary variables, he will be prompted to choose the atom for performing the unfolding step.

In the example below, the initial program is `rotate_leftdepth`, borrowed from [11]. The first clause of this program contains an unnecessary variable. By applying the uve strategy, we transform the `rotate_leftdepth` program into an equivalent program (w.r.t. the least Herbrand model semantics) without unnecessary variables.

```
| ?- st(rotate_leftdepth).

Initial program:
1. rotate_leftdepth(A,B) :- rotate(A,C), leftdepth(C,B).
2. rotate(leaf,leaf).
3. rotate(tree(A,B),tree(C,D)) :- rotate(A,C), rotate(B,D).
4. rotate(tree(A,B),tree(C,D)) :- rotate(A,D), rotate(B,C).
5. leftdepth(leaf,0).
6. leftdepth(tree(A,B),s(C)) :- leftdepth(A,C).

| ?- uve.
input clause number: 1
```

```
clause number among:
7.  new1(A,B) :- rotate(A,C), leftdepth(C,B).
|:  7.
atom Number:  1.

by unfolding clause 7 w.r.t. atom 1 we get:
8. new1(leaf,A) :- leftdepth(leaf,A).
9. new1(tree(A,B),C) :- rotate(A,D), rotate(B,E), leftdepth(tree(D,E),C).
10. new1(tree(A,B),C) :- rotate(A,D), rotate(B,E), leftdepth(tree(E,D),C).
by folding clause 9 w.r.t. atoms [1,2,3] using 11 we get:
12. new1(tree(A,B),C) :- new1(A,B,C).
by folding clause 10 w.r.t. atoms [2,1,3] using 11 we get:
13. new1(tree(A,B),C) :- new1(B,A,C).

clause number among:
11. new1(A,B,C) :- rotate(A,D), rotate(B,E), leftdepth(tree(D,E),C).

|:  11.
atom Number:  3.

by unfolding clause 11 w.r.t. atom 3 we get:
14. new1(A,B,s(C)) :- rotate(A,D), rotate(B,E), leftdepth(D,C).
by folding clause 14 w.r.t. atoms [1,3] using 7 we get:
16. new1(A,B,s(C)) :- new1(A,C), rotate(B,D).
by folding clause 16 w.r.t. atoms [2] using 15 we get:
17. new1(A,B,s(C)) :- new1(A,C), new1(B).

clause number among:
15. new1(A) :- rotate(A,B).

|:  15.
atom Number:  1.

new_program:
5. leftdepth(leaf,0).
6. leftdepth(tree(A,B),s(C)) :- leftdepth(A,C).
8. new1(leaf,A) :- leftdepth(leaf,A).
12. new1(tree(A,B),C) :- new1(A,B,C).
13. new1(tree(A,B),C) :- new1(B,A,C).
17. new1(A,B,s(C)) :- new1(A,C), new1(B).
18. new1(leaf).
22. new1(tree(A,B)) :- new1(A), new1(B).
24. new1(tree(A,B)) :- new1(A), new1(B).

| ?-
```

The derived program does not contain any unnecessary variable. It can be further simplified by unfolding clause 8, clause 12, and clause 13, and by deleting clause 24 which is subsumed by clause 22, and by eliminating the transient definition `new1/3`, as follows:

```
| ?-  u(8), ud_p(12), ud_p(13), subs.

by unfolding clause 8 w.r.t. atom 1 we get:
25. new1(leaf,0).
by unfolding clause 12 w.r.t. atom 1 using current program  we get:
26. new1(tree(A,B),s(C)) :- new1(A,C), new1(B).
```

```
by unfolding clause 13 w.r.t. atom 1 using current program  we get:
27. new1(tree(A,B),s(C)) :- new1(B,C), new1(A).
clause 22 subsumed by 24
| ?- restr(new1/2).
elimination of transient clause(s) [2,3,4,5,6,17]
| ?- p.
25. new1(leaf,0).
26. new1(tree(A,B),s(C)) :- new1(A,C), new1(B).
27. new1(tree(A,B),s(C)) :- new1(B,C), new1(A).
18. new1(leaf).
24. new1(tree(A,B)) :- new1(A), new1(B).
```

In the end, we get a final program made out of clauses 25, 26, 27, 18, and 24.

7.2 A strategy for loop absorption

The strategy `lap` implements the Loop Absorption strategy defined in [10]. To run the strategy, the user must first load an initial program via the command `start`, and then apply the command `lap`. Then he is asked whether some predicates of the initial program should be considered as *irrelevant*. This is to prevent the unfolding rule from being applied to a clause w.r.t. an atom with a irrelevant predicate. The user is also asked to give some values, namely the *horizontal bound* and the *vertical bound*, which are used in the strategy to control the size of the unfolding tree and to decide when the *generalization* rule must be applied. Details about the use and implementation of this strategy can be found in T. Morgante's thesis [7], which contains an algorithm, the code of the corresponding implementation, and several examples.
Below, we borrow an example from [7] to illustrate the use of the `lap` strategy. The initial program is the `nqueens` program.

```
| ?- st(queens).
Initial program:
1. queens(A,B) :- perm(A,C), pair(A,C,B), safe([],B).
2. perm([],[]).
3. perm([A|B],[C|D]) :- del(C,[A|B],E), perm(E,D).
4. del(A,[A|B],B).
5. del(A,[B|C],[B|D]) :- del(A,C,D).
6. pair([],[],[]).
7. pair([A|B],[C|D],[p(A,C)|E]) :- pair(B,D,E).
8. safe(A,[]).
9. safe(A,[B|C]) :- test(A,B), safe([B|A],C).
10. test([],A).
11. test([A|B],C) :- ndiag(A,C), test(B,C).
12. ndiag(p(A,B),p(C,D)) :- A-C=\=B-D, A-C=\=D-B.
| ?- lap.
Enter the clause number [1]:
predicates defined :
        ...perm/2
        ...del/3
        ...pair/3
        ...safe/2
        ...test/2
        ...ndiag/2
```

```
Do you want to mark the predicates (y/n) [n]?  y

Mark predicates with: return or r  if you want to mark that predicate,
                                i  if you do not want to mark that predicate,

Mark :perm/2 (r/i):  (r/i):  r
Mark :del/3 (r/i):  (r/i):  i
Mark :pair/3 (r/i):  (r/i):  r
Mark :safe/2 (r/i):  (r/i):  r
Mark :test/2 (r/i):  (r/i):  i
Mark :ndiag/2 (r/i):  (r/i):  i
Enter horizontal bound:  4.
Enter vertical bound:  4.

it all took 2.860 sec. Final program save in Final_Progs/lap_queens:

13. del(A,[A|B],B).
14. del(A,[B|C],[B|D]) :- del(A,C,D).
15. test([],A).
16. test([A|B],C) :- ndiag(A,C), test(B,C).
17. ndiag(p(A,B),p(C,D)) :- A-C=\=B-D, A-C=\=D-B.
18. new1([A|B],C,[D,E],p(E,F),p(D,G),H,[],I,J) :- del(C,[A|B],[K|L]),
        del(I,[K|L],[M|N]), del(G,[M|N],[O|P]), del(F,[O|P],[]).
19. new1([A|B],C,[D|E],F,p(D,G),H,[I|J],K,L) :- del(C,[A|B],M),
        new1(M,K,E,I,F,[L|H],J,G,p(D,G)), test([F,p(D,G),L|H],I).
20. queens([],[]).
21. queens([A],[p(A,B)]) :- del(B,[A],[]), test([],p(A,B)).
22. queens([A,B],[p(A,C),p(B,D)]) :- del(C,[A,B],[E|F]),
        del(D,[E|F],[]), test([p(A,C)],p(B,D)), test([],p(A,C)).
23. queens([A,B,C],[p(A,D),p(B,E),p(C,F)]) :- del(D,[A,B,C],[G|H]),
        del(E,[G|H],[I|J]), del(F,[I|J],[]), test([p(B,E),p(A,D)],p(C,F)),
        test([p(A,D)],p(B,E)), test([],p(A,D)).
24. queens([A,B,C,D],[p(A,E),p(B,F),p(C,G),p(D,H)]) :- del(E,[A,B,C,D],[I|J]),
        del(F,[I|J],[K|L]), del(G,[K|L],[M|N]), del(H,[M|N],[]),
        test([p(C,G),p(B,F),p(A,E)],p(D,H)), test([p(B,F),p(A,E)],p(C,G)),
        test([p(A,E)],p(B,F)), test([],p(A,E)).
25. queens([A,B,C|D],[p(A,E),p(B,F),p(C,G),H,I|J]) :- del(E,[A,B,C|D],K),
        new1(K,F,D,I,H,[p(B,F),p(A,E)],J,G,p(C,G)), test([H,p(C,G),p(B,F),
        p(A,E)],I), test([p(C,G),p(B,F),p(A,E)],H), test([p(B,F),p(A,E)],
        p(C,G)), test([p(A,E)],p(B,F)), test([],p(A,E)).
```

As we can see, the initial program first generates a possible configuration for the input list of queens, and then tests whether this configuration is safe or not. It is a naive *generate and test* algorithm. The final program however, is able to detect a failing configuration before it is completely specified, and therefore it is more efficient. It prevents from generating all possible permutations of the input list.

Several other examples of the use of the strategy `lap`, with various execution statistics can be found in [7].

7.3 A strategy for reducing nondeterminism

The strategy `determ` implements a strategy for reducing nondeterminism of programs based on an extension of the partial evaluation techniques, which has been formally presented in [9, 12]. One goal of the strategy is to produce a *semi-deterministic* program, that is, a program in which any two non-unit clauses have mutually exclusive instantiation of their head. Predicates *modes* are used to ensure that, at runtime, at most one clause head matches any given predicate call.

The strategy `determ` takes as parameters some subsidiary strategies which specify i) the application of each transformation rule, like the selection of atoms for unfolding (leftmost atom, leftmost *consumer*, deterministic unfolding), ii) whether or not and how case split must be applied, iii) whether or not and when clause subsumption must be applied, and iv) whether or not clauses with failing bodies must be deleted, etc. It can be seen as a while-do loop as depicted in Figure 2. Given a program P and a goal G w.r.t. we want to specialize P, we start off by introducing a set Def_1 consisting of a definition clause of the form $newp(X_1,\ldots,X_m) \leftarrow G$, where $X1,\ldots,X_m$ are the variables occurring in G. Then, from the program $P\cup\ Def_1$ we construct a sequence of programs of the form $P\cup\ Def_i \cup Q_i$, for $i \geq 1$, with $Q_1 = \{\}$. We construct the program $P\cup\ Def_{i+1} \cup Q_{i+1}$ from the program $P\cup\ Def_i \cup Q_i$ as follows. We first unfold all clauses in Def_i w.r.t. the leftmost atom of their bodies. Then we may either stop or continue unfolding the resulting clauses, depending on the parameters chosen for the unfolding strategie. Then for each predicate *newp* occurring in the set of clauses derived by unfolding, we constitute *packets*, with *mutually exclusive* heads, by applying zero or more times the case split rule. These applications of the case split rule are interleaved with applications of the subsumption rule and also with steps of simplification of inequalities whenever possible. These simplifications may replace an inequality by *true* or *false*. Clauses with occurrences of *false* in the body are deleted. Notice that after the applications of the case split and subsumption rules, packets containing unit clauses are singletons. Let U_i be the set of unit clauses and N_i be the set of packets each of them containing non-unit clauses only, derived from Def_i by the steps of unfolding, case split, subsumption, and simplification we have described. We then introduce a set Def_{i+1} of new definition clauses such that each packet of clauses in N_i can be folded using clauses in $Def_1 \cup \ldots \cup Def_{i+1}$ producing exactly one clause. We fold the clauses in N_i using the definition clauses belonging to $Def_1 \cup \ldots \cup Def_{i+1}$ and we derive a set F_i of mutually exclusive clauses and a new program $P \cup Def_{i+1} \cup Q_{i+1}$, where Q_{i+1} is $U_i \cup F_i \cup Q_i$.
To run the strategy, the user must apply the command `determ`. This will first consult the file `determ_strat_parameters.pl`. The initial program P and the initial set of definitions Def_1 must be specified in this file. Parameters for the subsidiary strategies, including a *mode* for every predicate of the initial program (see Section 7.3.2) must also be specified in this file before running the strategy, and also some other parameters, like for instance, the value of the variable `max_depth`, which prevents the transformation from being infinite, since there is no guarantee that the transformation process terminates. In Appendix C.17, we give two versions of this file which are used in the two examples presented in Section 7.3.1 and Section 7.3.3.

7.3.1 The string matching example

The strategy is run by issuing at top-level the command `determ` with no argument. Remember that the initial program and the specialized clause should have been already given in the file `determ_strat_parameters.pl`, with other parameters as explained above. In the example below, the initial program defines the predicate `match(P,S)` which specifies that the pattern `P` occurs in the string `S`. The specialized clause is `new1(S) :- match([a,a,b],S)`. This means, that we intend, via partial evaluation, to derive a program for the new predicate `new1(S)`, which specifies that a string `S` contains the pattern `aab`.

The derivation is constructed automatically and stops as soon as a deterministic program as been derived, or when the depth bound has been reached, or when an unexpected failure occurs. In the last case, the user may try to experiment with other input parameters, taking into account what has been derived before the failure occurred.
The derivation below is made of six iterations of the while-do-loop depicted in Figure 2, according to the parameters specified in the file `determ_strat_parameters.pl`. This file is given in Appendix C.17.1. For the sake of readibility of the derivation, some comments have been added to it, according to the notations used in Figure 2, at the end of each iteration. They use the normal font.

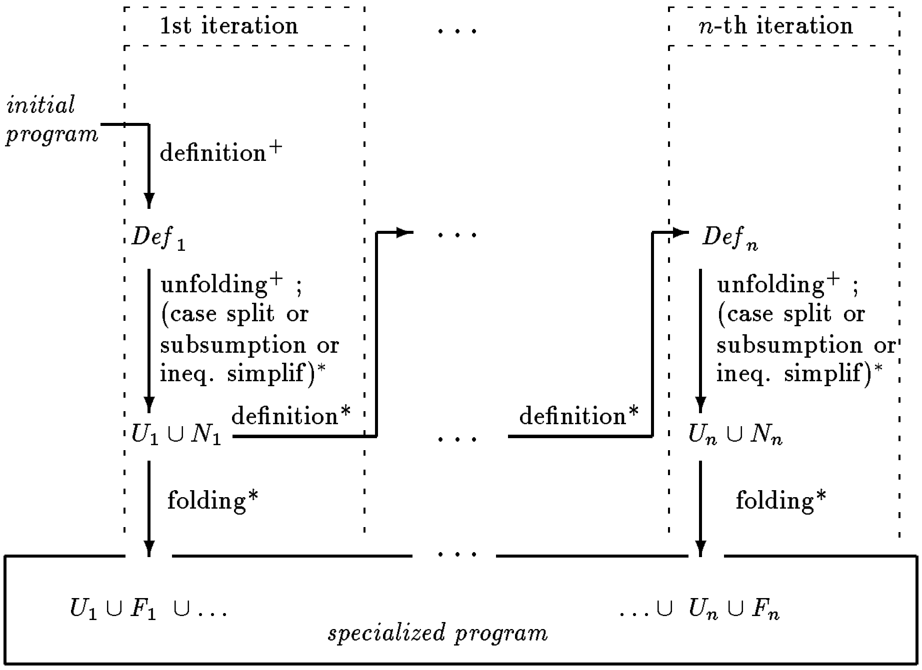

Figure 2: A strategy for reducing nondeterminism. r^+ denotes one or more application of the rule r, and r^* denotes zero or more application of r.

```
| ?- determ.

{consulting ~renault/Transf/determ_strat_parameters.pl...}
{~renault/Transf/determ_strat_parameters.pl consulted, 20 msec 944 bytes}
Strategy parameters:
apply_case_split(global_analysis)
apply_subsumption(after_unfolding)
after_first_step_unfold_leftmost_consumer
delete_clauses_with_failing_body
Strategy modes:
mode(match,2,[+,+])
mode(app,3,[-,-,+])
Initial program:
  1. match(A,B) :- app(C,D,B), app(E,A,C).
  2. app([],A,A).
  3. app([A|B],C,[A|D]) :- app(B,C,D).
Initial definition:
  4. new1(A) :- match([a,a,b],A).
by unfolding clause 4 w.r.t. atom 1 using current program  we get:
  5. new1(A) :- app(B,C,A), app(D,[a,a,b],B).
```

$P_1 = \{1, 2, 3\}, Def_1 = \{\{4\}\}, N_1 = \{\{5\}\}, U_1 = \{\}$

```
Introduction of definition new2:
  6. new2(A) :- app(B,C,A), app(D,[a,a,b],B).
by folding clause [5] using [6] we get:
  7. new1(A) :- new2(A).
by unfolding clause 6 w.r.t. atom 1 using current program  we get:
```

```
  8. new2(A) :- app(B,[a,a,b],[]).
  9. new2([A|B]) :- app(C,D,B), app(E,[a,a,b],[A|C]).
deletion of clause 8
by unfolding clause 9 w.r.t. atom 2 using current program  we get:
  10. new2([a|A]) :- app([a,b],B,A).
  11. new2([A|B]) :- app(C,D,B), app(E,[a,a,b],C).
by case splitting clauses [10,11] we get:
  12. new2([A|B]) :- \+A=a, app(C,D,B), app(E,[a,a,b],C).
  13. new2([a|A]) :- app(B,C,A), app(D,[a,a,b],B).
  14. new2([a|A]) :- app([a,b],B,A).
by folding clause [12] using [6] we get:
  15. new2([A|B]) :- \+A=a, new2(B).
```

$Def_2 = \{\{6\}\},\ F_1 = \{7, 15\},\ N_2 = \{\{13, 14\}\},\ U_2 = \{\}$

```
Introduction of definition new3:
  16. new3(A) :- app(B,C,A), app(D,[a,a,b],B).
  17. new3(A) :- app([a,b],B,A).
by folding clause [13,14] using [16,17] we get:
  18. new2([a|A]) :- new3(A).
by unfolding clause 16 w.r.t. atom 1 using current program  we get:
  19. new3(A) :- app(B,[a,a,b],[]).
  20. new3([A|B]) :- app(C,D,B), app(E,[a,a,b],[A|C]).
by unfolding clause 17 w.r.t. atom 1 using current program  we get:
  21. new3([a|A]) :- app([b],B,A).
deletion of clause 19
by unfolding clause 20 w.r.t. atom 2 using current program  we get:
  22. new3([a|A]) :- app([a,b],B,A).
  23. new3([A|B]) :- app(C,D,B), app(E,[a,a,b],C).
by case splitting clauses [22,23,21] we get:
  24. new3([A|B]) :- \+A=a, app(C,D,B), app(E,[a,a,b],C).
  25. new3([a|A]) :- app([b],B,A).
  26. new3([a|A]) :- app(B,C,A), app(D,[a,a,b],B).
  27. new3([a|A]) :- app([a,b],B,A).
by folding clause [24] using [6] we get:
  28. new3([A|B]) :- \+A=a, new2(B).
```

$Def_3 = \{\{16, 17\}\},\ F_2 = \{18, 28\},\ N_3 = \{\{25, 26, 27\}\},\ U_3 = \{\}$

```
Introduction of definition new4:
  29. new4(A) :- app([b],B,A).
  30. new4(A) :- app(B,C,A), app(D,[a,a,b],B).
  31. new4(A) :- app([a,b],B,A).
by folding clause [25,26,27] using [29,30,31] we get:
  32. new3([a|A]) :- new4(A).
by unfolding clause 29 w.r.t. atom 1 using current program  we get:
  33. new4([b|A]) :- app([],B,A).
by unfolding clause 30 w.r.t. atom 1 using current program  we get:
  34. new4(A) :- app(B,[a,a,b],[]).
  35. new4([A|B]) :- app(C,D,B), app(E,[a,a,b],[A|C]).
by unfolding clause 31 w.r.t. atom 1 using current program  we get:
  36. new4([a|A]) :- app([b],B,A).
deletion of clause 34
by unfolding clause 35 w.r.t. atom 2 using current program  we get:
```

```
  37. new4([a|A]) :- app([a,b],B,A).
  38. new4([A|B]) :- app(C,D,B), app(E,[a,a,b],C).
by case splitting clauses [33,37,38,36] we get:
  39. new4([A|B]) :- \+A=a, \+A=b, app(C,D,B), app(E,[a,a,b],C).
  40. new4([a|A]) :- app([b],B,A).
  41. new4([a|A]) :- app(B,C,A), app(D,[a,a,b],B).
  42. new4([a|A]) :- app([a,b],B,A).
  43. new4([b|A]) :- app(B,C,A), app(D,[a,a,b],B).
  44. new4([b|A]) :- app([],B,A).
by folding clause [39] using [6] we get:
45. new4([A|B]) :- \+A=a, \+A=b, new2(B).
by folding clause [40,41,42] using [29,30,31] we get:
  46. new4([a|A]) :- new4(A).
```

$Def_4 = \{\{29, 30, 31\}\}$, $F_3 = \{32, 45, 46\}$, $N_4 = \{\{43, 44\}\}$, $U_4 = \{\}$

```
Introduction of definition new5:
  47. new5(A) :- app(B,C,A), app(D,[a,a,b],B).
  48. new5(A) :- app([],B,A).
by folding clause [43,44] using [47,48] we get:
  49. new4([b|A]) :- new5(A).
by unfolding clause 47 w.r.t. atom 1 using current program  we get:
  50. new5(A) :- app(B,[a,a,b],[]).
  51. new5([A|B]) :- app(C,D,B), app(E,[a,a,b],[A|C]).
by unfolding clause 48 w.r.t. atom 1 using current program  we get:
  52. new5(A).
deletion of clause 50
by unfolding clause 51 w.r.t. atom 2 using current program  we get:
  53. new5([a|A]) :- app([a,b],B,A).
  54. new5([A|B]) :- app(C,D,B), app(E,[a,a,b],C).
clause 53 subsumed by 52
clause 54 subsumed by 52
```

$Def_5 = \{\{47, 48\}\}$, $F_4 = \{49\}$, $N_5 = \{\}$, $U_5 = \{52\}$

```
Final program :
7. new1(A) :- new2(A).
15. new2([A|B]) :- \+A=a, new2(B).
18. new2([a|A]) :- new3(A).
28. new3([A|B]) :- \+A=a, new2(B).
32. new3([a|A]) :- new4(A).
45. new4([A|B]) :- \+A=a, \+A=b, new2(B).
46. new4([a|A]) :- new4(A).
49. new4([b|A]) :- new5(A).
52. new5(A).
```

$Def_6 = \{\}$, $F_5 = \{\}$, and $P_{final} = P_1 \cup \bigcup_{i=1}^{5} U_i \cup F_i = \{7, 15, 18, 28, 32, 45, 46, 49, 52\}$.

The final program is semi-deterministic and the above derivation is an illustration of the power of the proposed strategy to produce efficient programs. Actually, it was able to derive from a general pattern matcher and a given pattern, a program which behaves like Knuth-Morris-Pratt algorithm. More details can be find in [9]. Notice that in [9], a different unfolding strategy has been used, which does not use the notion of modes, but in both cases, we derived the same final program.

Notice that the use of case split is not necessary, in this example, to produce a semi-deterministic program. By running the same example with the same parameters but without case split, we would have obtained the program below:

```
7. new1(A) :- new2(A).
14. new2([A|B]) :- new3(A,B).
23. new3(A,[B|C]) :- new4(A,B,C).
34. new4(A,B,[C|D]) :- new5(A,B,C,D).
35. new5(a,a,b,A).
42. new5(A,B,C,[D|E]) :- new5(B,C,D,E).
```

This program is still semi-deterministic in the sense of [12], but it does not behave like the Knuth-Morris-Pratt algorithm, because it has *look-ahead* greater than one, in the sense that more than one input letter of the pattern is taken into account during the head unification process (see clause 35.).

7.3.2 Some remarks about predicate modes and the case split rule

The strategy illustrated above uses the notion of predicate *modes*. A *mode* is given by associating '+' or '-' for each argument of a predicate. Intuitively '+' corresponds to an argument which is ground at runtime. We will call these arguments *input* arguments.

Modes are used to apply the case split rule in order to choose the arguments w.r.t. which case split must be performed, so as to produce semi-deterministic programs.

Modes may also be used in the unfolding strategy, via the notion of *consumer* or the notion of *weak consumer*. A consumer (resp. a weak consumer) is an atom in which every (resp. at least one) input argument contains a non variables term.

In the example of Section 7.3.1, the parameter `after_first_step_unfold_leftmost_consumer` means that, after having unfolded once a definition clause (and this unfolding is performed w.r.t. the leftmost atom), we continue by unfolding all the resulting clauses as long as they contain in their bodies an atom which is a consumer.

We have implemented two variants for the case split rule.

1. binary case split applies to a pair of clause C_1 and C_2 and replace C_1 by two equivalent mutually exclusive clauses C_{11} and C_{12} so that C_{12} and C_2 are also mutually exclusive. Formally, let C_1 and C_2 be as follows:
   $C_1:\ p(t_1,\ldots,X_i,\ldots,t_n) \leftarrow Bd_1$
   $C_2:\ p(u_1,\ldots,u_i,\ldots,u_n) \leftarrow Bd_2$
   such that i is the smallest integer which verifies the following conditions:
   (a) the mode of predicate p for its i-th argument is '+'.
   (b) X_i is a variable term and u_i is a non variable term

   then C_{11} and C_{12} are:
   $C_{11}:\ (p(t_1,\ldots,X_i,\ldots,t_n) \leftarrow Bd_1)\{X_i/u_i\}$
   $C_{12}:\ p(t_1,\ldots,X_i,\ldots,t_n) \leftarrow X_i \neq u_i,\ Bd_1$

   Two clauses are mutually exclusive if they have no instance with the same arguments at every input position, or if there is an inequality in (the body of) one of the two instances which fails. In the above example, the instance of C_{12} which makes its head equal to the one of C_{11} has a failing inequality in its body ($u_i \neq u_i$).

   Notice that in this application of case split, the head of C_{11} is not given explicitly as it was the case for the case split rule described in Section 6.1, but it is inferred by comparing the input arguments of the heads of C_1 and C_2.

2. case split by 'global analysis' applies to a set of clauses and produces in one step a set of sets of clauses called *packets*, such that in each packet every clauses have the same input arguments in their heads, and every two clauses from two different packets are mutually exclusive.

Case split by 'global analysis' has been used in the example of Section 7.3.1, while binary case split is used in the example of Section 7.3.3. It turns out that the program derived in Section 7.3.3 by using binary case split is much shorter than the one which is obtained if we use case split by 'global analysis' instead.
The use of one version of the case split rule instead of the other one, may produce different programs, because the order in which the packets are consituted, and therefore the order in which new definitions are introduced may differ. A possible application of subsumption after one application of the binary case split rule may also remove clauses while the whole packets are not completely constituted.

7.3.3 The multi-string matching example

The example below is borrowed from [12]. The initial program defines the predicate `m_match(Ps,S,N)` which specifies that there exists a pattern in the list `Ps` of patterns which occurs in the string `S` at position `N`. The specialized clause is `new1(S,N) :- m_match([[a,a,a],[a,a,b]],S,N)`. This means, that we intend to derive a program for the new predicate `new1(S,N)`, which specifies that the string `S` contains the pattern `aaa` or the pattern `aab` at position N. The file `determ_strat_parameters.pl` used for this derivation is given in Appendix C.17.2.

```
  | ?- determ.

{consulting ~renault/Transf/determ_strat_parameters.pl...}
{~renault/Transf/determ_strat_parameters.pl consulted, 20 msec 752 bytes}
Strategy parameters:
apply_case_split(binary)
delete_clauses_with_failing_body
after_first_step_unfold_leftmost_weak_consumer
Strategy modes:
mode(m_match,3,[+,+,-])
mode(match,3,[+,+,-])
mode(app,3,[-,-,+])
mode(len,2,[+,-])
Initial program:
1. m_match([A|B],C,D) :- match(A,C,D).
2. m_match([A|B],C,D) :- m_match(B,C,D).
3. match(A,B,C) :- app(D,E,B), app(F,A,D), len(F,C).
4. len([],0).
5. len([A|B],s(C)) :- len(B,C).
6. app([],A,A).
7. app([A|B],C,[A|D]) :- app(B,C,D).
Initial definition:
8. new1(A,B) :- m_match([[a,a,a],[a,a,b]],A,B).
by unfolding clause 8 w.r.t. atom 1 using current program  we get:
9. new1(A,B) :- match([a,a,a],A,B).
10. new1(A,B) :- m_match([[a,a,b]],A,B).
by unfolding clause 9 w.r.t. atom 1 using current program  we get:
11. new1(A,B) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
by unfolding clause 10 w.r.t. atom 1 using current program  we get:
12. new1(A,B) :- match([a,a,b],A,B).
```

```
13. new1(A,B) :- m_match([],A,B).
by unfolding clause 12 w.r.t. atom 1 using current program  we get:
14. new1(A,B) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
deletion of clause 13
```

$P_1 = \{1,2,3,4,5,6,7\}, Def_1 = \{\{8\}\}, N_1 = \{\{11,14\}\}, U_1 = \{\}$

```
Introduction of definition new2:
15. new2(A,B) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
16. new2(A,B) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
by folding clause [11,14] using [15,16] we get:
17. new1(A,B) :- new2(A,B).
by unfolding clause 15 w.r.t. atom 1 using current program  we get:
18. new2(A,B) :- app(C,[a,a,a],[]), len(C,B).
19. new2([A|B],C) :- app(D,E,B), app(F,[a,a,a],[A|D]), len(F,C).
by unfolding clause 16 w.r.t. atom 1 using current program  we get:
20. new2(A,B) :- app(C,[a,a,b],[]), len(C,B).
21. new2([A|B],C) :- app(D,E,B), app(F,[a,a,b],[A|D]), len(F,C).
deletion of clause 18
by unfolding clause 19 w.r.t. atom 2 using current program  we get:
22. new2([a|A],B) :- app([a,a],C,A), len([],B).
23. new2([A|B],C) :- app(D,E,B), app(F,[a,a,a],D), len([A|F],C).
by unfolding clause 22 w.r.t. atom 2 using current program  we get:
24. new2([a|A],0) :- app([a,a],B,A).
by unfolding clause 23 w.r.t. atom 3 using current program  we get:
25. new2([A|B],s(C)) :- app(D,E,B), app(F,[a,a,a],D), len(F,C).
deletion of clause 20
by unfolding clause 21 w.r.t. atom 2 using current program  we get:
26. new2([a|A],B) :- app([a,b],C,A), len([],B).
27. new2([A|B],C) :- app(D,E,B), app(F,[a,a,b],D), len([A|F],C).
by unfolding clause 26 w.r.t. atom 2 using current program  we get:
28. new2([a|A],0) :- app([a,b],B,A).
by unfolding clause 27 w.r.t. atom 3 using current program  we get:
29. new2([A|B],s(C)) :- app(D,E,B), app(F,[a,a,b],D), len(F,C).
by case splitting clause 25 w.r.t. clause 24 we get:
30. new2([a|A],s(B)) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
31. new2([A|B],s(C)) :- \+A=a, app(D,E,B), app(F,[a,a,a],D), len(F,C).
by case splitting clause 29 w.r.t. clause 24 we get:
32. new2([a|A],s(B)) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
33. new2([A|B],s(C)) :- \+A=a, app(D,E,B), app(F,[a,a,b],D), len(F,C).
```

$Def_2 = \{\{15,16\}\},\ F_1 = \{17\},\ N_2 = \{\{24,28,30,32\},\{31,33\}\},\ U_2 = \{\}$

```
Introduction of definition new3:
34. new3(A,0) :- app([a,a],B,A).
35. new3(A,s(B)) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
36. new3(A,0) :- app([a,b],B,A).
37. new3(A,s(B)) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
by folding clause [24,30,28,32] using [34,35,36,37] we get:
38. new2([a|A],B) :- new3(A,B).
by folding clause [31,33] using [15,16] we get:
39. new2([A|B],s(C)) :- \+A=a, new2(B,C).
by unfolding clause 34 w.r.t. atom 1 using current program  we get:
40. new3([a|A],0) :- app([a],B,A).
```

```
by unfolding clause 35 w.r.t. atom 1 using current program  we get:
41. new3(A,s(B)) :- app(C,[a,a,a],[]), len(C,B).
42. new3([A|B],s(C)) :- app(D,E,B), app(F,[a,a,a],[A|D]), len(F,C).
by unfolding clause 36 w.r.t. atom 1 using current program  we get:
43. new3([a|A],0) :- app([b],B,A).
by unfolding clause 37 w.r.t. atom 1 using current program  we get:
44. new3(A,s(B)) :- app(C,[a,a,b],[]), len(C,B).
45. new3([A|B],s(C)) :- app(D,E,B), app(F,[a,a,b],[A|D]), len(F,C).
deletion of clause 41
by unfolding clause 42 w.r.t. atom 2 using current program  we get:
46. new3([a|A],s(B)) :- app([a,a],C,A), len([],B).
47. new3([A|B],s(C)) :- app(D,E,B), app(F,[a,a,a],D), len([A|F],C).
by unfolding clause 46 w.r.t. atom 2 using current program  we get:
48. new3([a|A],s(0)) :- app([a,a],B,A).
by unfolding clause 47 w.r.t. atom 3 using current program  we get:
49. new3([A|B],s(s(C))) :- app(D,E,B), app(F,[a,a,a],D), len(F,C).
deletion of clause 44
by unfolding clause 45 w.r.t. atom 2 using current program  we get:
50. new3([a|A],s(B)) :- app([a,b],C,A), len([],B).
51. new3([A|B],s(C)) :- app(D,E,B), app(F,[a,a,b],D), len([A|F],C).
by unfolding clause 50 w.r.t. atom 2 using current program  we get:
52. new3([a|A],s(0)) :- app([a,b],B,A).
by unfolding clause 51 w.r.t. atom 3 using current program  we get:
53. new3([A|B],s(s(C))) :- app(D,E,B), app(F,[a,a,b],D), len(F,C).
by case splitting clause 49 w.r.t. clause 40 we get:
54. new3([a|A],s(s(B))) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
55. new3([A|B],s(s(C))) :- \+A=a, app(D,E,B), app(F,[a,a,a],D), len(F,C).
by case splitting clause 53 w.r.t. clause 40 we get:
56. new3([a|A],s(s(B))) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
57. new3([A|B],s(s(C))) :- \+A=a, app(D,E,B), app(F,[a,a,b],D), len(F,C).
```

$Def_3 = \{\{34, 35, 36, 37\}\},\ F_2 = \{38, 39\},\ N_3 = \{\{43, 48, 52, 54, 56\}, \{55, 57\}\},\ U_3 = \{\}$

```
Introduction of definition new4:
58. new4(A,0) :- app([a],B,A).
59. new4(A,s(0)) :- app([a,a],B,A).
60. new4(A,s(s(B))) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
61. new4(A,0) :- app([b],B,A).
62. new4(A,s(0)) :- app([a,b],B,A).
63. new4(A,s(s(B))) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
by folding clause [40,48,54,43,52,56] using [58,59,60,61,62,63] we get:
64. new3([a|A],B) :- new4(A,B).
by folding clause [55,57] using [15,16] we get:
65. new3([A|B],s(s(C))) :- \+A=a, new2(B,C).
by unfolding clause 58 w.r.t. atom 1 using current program  we get:
66. new4([a|A],0) :- app([],B,A).
by unfolding clause 59 w.r.t. atom 1 using current program  we get:
67. new4([a|A],s(0)) :- app([a],B,A).
by unfolding clause 60 w.r.t. atom 1 using current program  we get:
68. new4(A,s(s(B))) :- app(C,[a,a,a],[]), len(C,B).
69. new4([A|B],s(s(C))) :- app(D,E,B), app(F,[a,a,a],[A|D]), len(F,C).
by unfolding clause 61 w.r.t. atom 1 using current program  we get:
70. new4([b|A],0) :- app([],B,A).
```

```
by unfolding clause 62 w.r.t. atom 1 using current program  we get:
71. new4([a|A],s(0)) :- app([b],B,A).
by unfolding clause 63 w.r.t. atom 1 using current program  we get:
72. new4(A,s(s(B))) :- app(C,[a,a,b],[]), len(C,B).
73. new4([A|B],s(s(C))) :- app(D,E,B), app(F,[a,a,b],[A|D]), len(F,C).
deletion of clause 68
by unfolding clause 69 w.r.t. atom 2 using current program  we get:
74. new4([a|A],s(s(B))) :- app([a,a],C,A), len([],B).
75. new4([A|B],s(s(C))) :- app(D,E,B), app(F,[a,a,a],D), len([A|F],C).
by unfolding clause 74 w.r.t. atom 2 using current program  we get:
76. new4([a|A],s(s(0))) :- app([a,a],B,A).
by unfolding clause 75 w.r.t. atom 3 using current program  we get:
77. new4([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,a],D), len(F,C).
deletion of clause 72
by unfolding clause 73 w.r.t. atom 2 using current program  we get:
78. new4([a|A],s(s(B))) :- app([a,b],C,A), len([],B).
79. new4([A|B],s(s(C))) :- app(D,E,B), app(F,[a,a,b],D), len([A|F],C).
by unfolding clause 78 w.r.t. atom 2 using current program  we get:
80. new4([a|A],s(s(0))) :- app([a,b],B,A).
by unfolding clause 79 w.r.t. atom 3 using current program  we get:
81. new4([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,b],D), len(F,C).
by case splitting clause 77 w.r.t. clause 66 we get:
82. new4([a|A],s(s(s(B)))) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
83. new4([A|B],s(s(s(C)))) :- \+A=a, app(D,E,B), app(F,[a,a,a],D), len(F,C).
by case splitting clause 81 w.r.t. clause 66 we get:
84. new4([a|A],s(s(s(B)))) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
85. new4([A|B],s(s(s(C)))) :- \+A=a, app(D,E,B), app(F,[a,a,b],D), len(F,C).
by case splitting clause 83 w.r.t. clause 70 we get:
86. new4([b|A],s(s(s(B)))) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
87. new4([A|B],s(s(s(C)))) :- \+A=b, \+A=a, app(D,E,B), app(F,[a,a,a],D),
          len(F,C).
by case splitting clause 85 w.r.t. clause 70 we get:
88. new4([b|A],s(s(s(B)))) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
89. new4([A|B],s(s(s(C)))) :- \+A=b, \+A=a, app(D,E,B), app(F,[a,a,b],D),
          len(F,C).
```

$Def_4 = \{\{58, 59, 60, 61, 62, 63\}\}$, $F_3 = \{64, 65\}$, $N_4 = \{\{66, 67, 71, 76, 80, 82, 84\}, \{70, 86, 88\}, \{87, 89\}\}$, $U_4 = \{\}$

```
Introduction of definition new5:
90. new5(A,0) :- app([],B,A).
91. new5(A,s(0)) :- app([a],B,A).
92. new5(A,s(s(0))) :- app([a,a],B,A).
93. new5(A,s(s(s(B)))) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
94. new5(A,s(0)) :- app([b],B,A).
95. new5(A,s(s(0))) :- app([a,b],B,A).
96. new5(A,s(s(s(B)))) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
by folding clause [66,67,76,82,71,80,84] using [90,91,92,93,94,95,96] we get:
97. new4([a|A],B) :- new5(A,B).

Introduction of definition new6:
98. new6(A,s(s(s(B)))) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
99. new6(A,0) :- app([],B,A).
```

```
100. new6(A,s(s(s(B)))) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
by folding clause [86,70,88] using [98,99,100] we get:
101. new4([b|A],B) :- new6(A,B).
by folding clause [87,89] using [15,16] we get:
102. new4([A|B],s(s(s(C)))) :- \+A=b, \+A=a, new2(B,C).
by unfolding clause 90 w.r.t. atom 1 using current program  we get:
103. new5(A,0).
by unfolding clause 91 w.r.t. atom 1 using current program  we get:
104. new5([a|A],s(0)) :- app([],B,A).
by unfolding clause 92 w.r.t. atom 1 using current program  we get:
105. new5([a|A],s(s(0))) :- app([a],B,A).
by unfolding clause 93 w.r.t. atom 1 using current program  we get:
106. new5(A,s(s(s(B)))) :- app(C,[a,a,a],[]), len(C,B).
107. new5([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,a],[A|D]), len(F,C).
by unfolding clause 94 w.r.t. atom 1 using current program  we get:
108. new5([b|A],s(0)) :- app([],B,A).
by unfolding clause 95 w.r.t. atom 1 using current program  we get:
109. new5([a|A],s(s(0))) :- app([b],B,A).
by unfolding clause 96 w.r.t. atom 1 using current program  we get:
110. new5(A,s(s(s(B)))) :- app(C,[a,a,b],[]), len(C,B).
111. new5([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,b],[A|D]), len(F,C).
deletion of clause 106
by unfolding clause 107 w.r.t. atom 2 using current program  we get:
112. new5([a|A],s(s(s(B)))) :- app([a,a],C,A), len([],B).
113. new5([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,a],D), len([A|F],C).
by unfolding clause 112 w.r.t. atom 2 using current program  we get:
114. new5([a|A],s(s(s(0)))) :- app([a,a],B,A).
by unfolding clause 113 w.r.t. atom 3 using current program  we get:
115. new5([A|B],s(s(s(s(C))))) :- app(D,E,B), app(F,[a,a,a],D), len(F,C).
deletion of clause 110
by unfolding clause 111 w.r.t. atom 2 using current program  we get:
116. new5([a|A],s(s(s(B)))) :- app([a,b],C,A), len([],B).
117. new5([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,b],D), len([A|F],C).
by unfolding clause 116 w.r.t. atom 2 using current program  we get:
118. new5([a|A],s(s(s(0)))) :- app([a,b],B,A).
by unfolding clause 117 w.r.t. atom 3 using current program  we get:
119. new5([A|B],s(s(s(s(C))))) :- app(D,E,B), app(F,[a,a,b],D), len(F,C).
by case splitting clause 115 w.r.t. clause 104 we get:
120. new5([a|A],s(s(s(s(B))))) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
121. new5([A|B],s(s(s(s(C))))) :- \+A=a, app(D,E,B), app(F,[a,a,a],D), len(F,C).
by case splitting clause 119 w.r.t. clause 104 we get:
122. new5([a|A],s(s(s(s(B))))) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
123. new5([A|B],s(s(s(s(C))))) :- \+A=a, app(D,E,B), app(F,[a,a,b],D), len(F,C).
by case splitting clause 121 w.r.t. clause 108 we get:
124. new5([b|A],s(s(s(s(B))))) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
125. new5([A|B],s(s(s(s(C))))) :- \+A=b, \+A=a, app(D,E,B), app(F,[a,a,a],D),
          len(F,C).
by case splitting clause 123 w.r.t. clause 108 we get:
126. new5([b|A],s(s(s(s(B))))) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
127. new5([A|B],s(s(s(s(C))))) :- \+A=b, \+A=a, app(D,E,B), app(F,[a,a,b],D),
          len(F,C).
by folding clause [104,105,114,120,109,118,122] using [90,91,92,93,94,95,96] we get:
```

```
128. new5([a|A],s(B)) :- new5(A,B).
by folding clause [124,108,126] using [98,99,100] we get:
129. new5([b|A],s(B)) :- new6(A,B).
by folding clause [125,127] using [15,16] we get:
130. new5([A|B],s(s(s(s(C))))) :- \+A=b, \+A=a, new2(B,C).
by unfolding clause 98 w.r.t. atom 1 using current program  we get:
131. new6(A,s(s(s(B)))) :- app(C,[a,a,a],[]), len(C,B).
132. new6([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,a],[A|D]), len(F,C).
by unfolding clause 99 w.r.t. atom 1 using current program  we get:
133. new6(A,0).
by unfolding clause 100 w.r.t. atom 1 using current program  we get:
134. new6(A,s(s(s(B)))) :- app(C,[a,a,b],[]), len(C,B).
135. new6([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,b],[A|D]), len(F,C).
deletion of clause 131
by unfolding clause 132 w.r.t. atom 2 using current program  we get:
136. new6([a|A],s(s(s(B)))) :- app([a,a],C,A), len([],B).
137. new6([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,a],D), len([A|F],C).
by unfolding clause 136 w.r.t. atom 2 using current program  we get:
138. new6([a|A],s(s(s(0)))) :- app([a,a],B,A).
by unfolding clause 137 w.r.t. atom 3 using current program  we get:
139. new6([A|B],s(s(s(s(C))))) :- app(D,E,B), app(F,[a,a,a],D), len(F,C).
deletion of clause 134
by unfolding clause 135 w.r.t. atom 2 using current program  we get:
140. new6([a|A],s(s(s(B)))) :- app([a,b],C,A), len([],B).
141. new6([A|B],s(s(s(C)))) :- app(D,E,B), app(F,[a,a,b],D), len([A|F],C).
by unfolding clause 140 w.r.t. atom 2 using current program  we get:
142. new6([a|A],s(s(s(0)))) :- app([a,b],B,A).
by unfolding clause 141 w.r.t. atom 3 using current program  we get:
143. new6([A|B],s(s(s(s(C))))) :- app(D,E,B), app(F,[a,a,b],D), len(F,C).
by case splitting clause 139 w.r.t. clause 138 we get:
144. new6([a|A],s(s(s(s(B))))) :- app(C,D,A), app(E,[a,a,a],C), len(E,B).
145. new6([A|B],s(s(s(s(C))))) :- \+A=a, app(D,E,B), app(F,[a,a,a],D), len(F,C).
by case splitting clause 143 w.r.t. clause 138 we get:
146. new6([a|A],s(s(s(s(B))))) :- app(C,D,A), app(E,[a,a,b],C), len(E,B).
147. new6([A|B],s(s(s(s(C))))) :- \+A=a, app(D,E,B), app(F,[a,a,b],D), len(F,C).
by folding clause [138,144,142,146] using [34,35,36,37] we get:
148. new6([a|A],s(s(s(B)))) :- new3(A,B).
by folding clause [145,147] using [15,16] we get:
149. new6([A|B],s(s(s(s(C))))) :- \+A=a, new2(B,C).
```

$Def_5 = \{\{90, 91, 92, 93, 94, 95, 96\}, \{98, 99, 100\}\}$, $F_4 = \{97, 101, 102, 128, 129, 130, 148, 149\}$, $N_5 = \{\}$, $U_5 = \{103, 133\}$

```
Final program :
17. new1(A,B) :- new2(A,B).
38. new2([a|A],B) :- new3(A,B).
39. new2([A|B],s(C)) :- \+A=a, new2(B,C).
64. new3([a|A],B) :- new4(A,B).
65. new3([A|B],s(s(C))) :- \+A=a, new2(B,C).
97. new4([a|A],B) :- new5(A,B).
101. new4([b|A],B) :- new6(A,B).
102. new4([A|B],s(s(s(C)))) :- \+A=b, \+A=a, new2(B,C).
103. new5(A,0).
```

```
128. new5([a|A],s(B)) :- new5(A,B).
129. new5([b|A],s(B)) :- new6(A,B).
130. new5([A|B],s(s(s(s(C))))) :- \+A=b, \+A=a, new2(B,C).
133. new6(A,0).
148. new6([a|A],s(s(s(B)))) :- new3(A,B).
149. new6([A|B],s(s(s(s(C))))) :- \+A=a, new2(B,C).
```

$Def_6 = \{\}$, $F_5 = \{\}$, and
$P_{final} = P_1 \cup \bigcup_{i=1}^{5} U_i \cup F_i = \{17, 38, 39, 64, 65, 97, 101, 102, 103, 128, 129, 130, 133, 148, 149\}$.

Remark: The strategy used in the above derivation is slightly different from the one given in [12]. In [12], head generalization is performed (at each iteration) before folding a packet of clauses, and as a consequence, equalities occur in the clauses of the new definitions introduced for folding, whereas in the above derivation, equalitites never appear explicitly. However, the folded clauses are the same in both cases, and the final program above derived is the same as the one obtained in [12]. The head generalization step is necessary in general to ensure the soundness of the inference of predicate modes, however it can be safely omitted in this particular example.

8 Conclusions and future work

We have presented a tool for transforming logic programs which allows for both interactive and fully automatic program transformation. The possibilities offered to the user are limited to a predefined set of transformation rules and strategies. The strategy illustrated in Section 7.3 which allows us to reduce nondeterminism in programs, is parametric w.r.t. subsidiary strategies related to the application of the unfolding, case split, and subsumption rules. By choosing particular instances of these subsidiary strategies, the user may apply the general strategy in various ways by choosing, for instance, whether or not to apply case split and whether or not to apply subsumption, and if so, after which rule. Since the strategy might terminate for some values of the parameters and not terminate for some other values, it is useful that the user has the possibility to modify them. However, his possibilities in setting parameters are limited to predefined values, and they cannot be changed during a derivation. The use of the definition introduction rule and the folding rule in this strategy is fixed, namely, a definition body always corresponds to a clause body and therefore it is impossible to fold a clause w.r.t. a part of its body only. Moreover, the user does not have the possibility to add a new transformation rule nor to define a new strategy. In order to allow for more flexibility of the system and its extendability, a better system architecture must be defined. Future work will focus on this point and, in particular, we will deal with the internal representation of the transformation rules and strategies to allow for generality and language independency. The system should support the introduction of new transformation rules and the definition of new strategies in a dynamic way. This would require the definition of a user-oriented meta-language for the specification of rules and strategies. Another issue is the development of a graphical user interface in order to facilitate user's experimentations with the transformer. Some work has already been done in this direction in [5].

9 Acknowledgments

I am grateful to Alberto Pettorossi and Maurizio Proietti for their guidance, for the interest they have always shown in the development and in the use of the system, and for encouraging me in the continuation of this project. I thank them also for their careful reading and comments to improve earlier drafts of this report. The work has been realized at the University of Rome Tor Vergata and at IASI-CNR. I have been supported by the EC under the Human Capital and Mobility Project "Logic Program Synthesis and Transformation".

A Guidelines to the use of the transformer

In this appendix we give the basic instructions to install and run the transformer and the list of top-level commands for quick reference.

Installation

To install the SICStus (resp. Quintus) version of the transformer one needs the package `t6.tar.gz` (resp. `t5.tar.gz`) to be uncompressed and unpacked as usual. For instance for the SICStus package, one must type the following commands:

```
> gunzip t6.tar.gz
> tar xvf t6.tar
```

This will create a directory `TRANSFORMER/` containing subdirectories and files as described in Section 5 of the user's manual.

Invoking the transformer

SICStus (resp. Quintus) Prolog is needed to run the transformer. Once the transformer has been installed, it can be run from the Unix command level. The user must be in the directory `TRANSFORMER/` and type the command `t6` (resp. `t5`) at the Unix prompt, which will output a menu and prompt the user as follow:

```
     st.            < start a new transformation process >
     r.             < show available rules >
     s.             < show available strategies >
     c.             < show available other commands >
     helps.         < show this five lines menu >
| ?-
```

Prior to start a transformation process, the user should save the initial program in a file, say file `foo` in subdirectory `Progs/`. This program must be written in usual Prolog syntax. Then, he can start a derivation of program `foo` as follows:

```
| ?- st(foo).
```

which will give a number and show to the user each clause of file `foo`. Then, the user may start to issue some commands for transforming program `foo`. All available commands are summarized in Section A (See also Section 6 of the user's manual).

Exiting from the transformer

To exit from the transformer, one must use the `halt` Prolog goal, as follows:

```
| ?- halt.
```

Summary of top-level commands

Definition introduction rules

`ts_d`	Tamaki and Sato's definition introduction rule (one clause only).
`d`	definition introduction rule (possibly several clauses).

Unfolding rules

`u(C,A)`	unfold clause `C` wrt atom `A` using initial program. clause deletion is not performed if no head of clause unifies with `A`.
`u(C)`	short for `u(C,1)`.
`ud_i(Pi,C,A)`	unfold clause `C` wrt atom `A` using program `Pi`. clause deletion if the predicate of `A` occurs in the head of a clause in `Pi` and `A` does not unify with the head of any clause in `Pi`.
`ud_i(Pi,C)`	short for `ud_i(Pi,C,1)`.
`ud(C,A)`	short for `ud_i(1,C,1)`.
`ud(C)`	short for `ud(C,1)`.
`ud_p(C,A)`	short for `ud_i(Pn,C,A)` where `Pn` is the current program.
`ud_p(C)`	short for `ud_p(C,1)`.
`ud_facts(C)`	unfold clause `C` w.r.t atoms which are instances of unit clauses.
`ud_facts`	applies `ud_facts(C)` to every clause `C` in current program.
`ud_eq(C)`	unfold clause `C` w.r.t equalities.
`ud_eq`	applies `ud_eq(C)`, to every clause `C` in current program.

Folding rules

`ts_f(C,As,D)`	fold clause `C` using definition `D`, w.r.t. atoms in list `As` (Tamaki and Sato).
`f(Cs1,Cs2)`	fold clauses `Cs1` using clauses `Cs2`.
`r_f(Cs1,Cs2)`	same as `f(Cs1,Cs2)` but allow for permutations in lists `Cs1` and `Cs2`.

Goal replacement rules

`rep(C,As,L)`	replacement of goal made out of atoms `As` in clause `C` using law `L`.
`mv(C,J1,J2)`	move atom from position `J1` to position `J2` in the body of clause `C`.
`add_g(C)`	add a goal in the body of clause `C`.
`del_g(C,As)`	delete goal made out of atoms `As` in the body of clause `C`.

Clause replacement rules

`restr(F/A)`	delete clauses that are not used in the definition of predicate `F` of arity `A`.
`del_cl(Cs)`	delete clause(s) `Cs`.
`del_f_cl(Cs)`	delete among clause(s) `Cs` those which have a failing body.
`del_f_cl`	short for `del_f_cl(Cs)`, where `Cs` is the whole clauses in current program.
`add_cl`	add a clause to current program.
`rep_cl(C)`	replace clause `C` in current program by another clause.
`mv_cl(C)`	move clause `C` in current program to a new position.
`subs(Cs)`	apply subsumption to clauses `Cs`.
`subs`	short for `subs(Cs)`, where `Cs` is the list of all clauses in current program.
`split(C,A)`	split clause `C` w.r.t. atom `A`.

Commands for starting or navigation the transformation tree

`st(F)`	start a transformation with `F` as initial program.
`st`	same as `st(F)`, except that `F` is selected via a menu.
`load_s`	restore a transformation tree.
`set(P)`	move to program `P`.

Commands for selecting or editing replacement laws

`add_l(T)`	add a law to theory `T`.
`add_l`	add a law to current theory.
`del_l(T)`	delete a law from theory `T`.
`del_l`	delete a law from current theory.
`sel_l`	select laws from a theory and add them to current program.

Commands for showing information

`p1`	show initial program.
`p`	show current program.
`p(Ps)`	show program(s) `Ps`.
`pp`	same as `p`, but only clauses that are not also in initial program are shown.
`pp(P)`	same as `p(P)`, but only clauses that are not also in initial program are shown.
`h`	show history of current program.
`h(P)`	show history of program P.
`cls(Cs)`	show clauses `Cs`.
`defs`	show definitions introduced so far.
`l`	show laws present in current theory.
`dir_p`	show files contained in directory `Progs/`.
`dir_t`	show files contained in directory `Theories/`.
`dir_s`	show files contained in directory `Sessions/`.
`dir_f`	show files contained in directory `Final_Progs/`.

Commands for saving information

`s_p(P)`	save program `P` in directory `Final_Progs/`.
`s_p`	save current program in directory `Final_Progs/`.
`s_h(P)`	save history of program `P` in directory `Histories/`.
`s_h`	save history of current program in directory `Histories/`.
`s_th`	save current history in directory `Theories/`.
`s_der`	save current derivation in directory `Sessions/`.

Commands for adding comments

`com_h`	add comments to current history.

Commands for getting help

`helps`	show a general help menu.
`r`	show a menu of transformation rules.
`s`	show a menu of strategies.
`c`	show a menu of other commands.
`man(X)`	give help about the use of rule, command, or strategy `X`.

Strategies

`uve`	unnecessary variable elimination strategy.
`lap`	loop absorption strategy.
`determ`	reduction of nondeterminism (via specialization) strategy.

B Summary of help messages associated with top-level commands

This appendix contains all messages that can be shown to the user by running the command `man(X)`. `X` must be the name, or the shortname, of a top-level command presented in Section 6. To obtain the messages below, we have run `man(ts_d)`, `man(d)`, `man(u)`, `man(ud_i)`, etc. Notice that `X` must be given without any argument. For instance, we must run `man(u)` and not `man(u(C,A)`.

Definition introduction rules

• `ts_d`: Adds a new definition (using one clause only).

```
Example: | ?- ts_d.
         enter definition clause:
         |: r :- q.
         3. r :- q.
         | ?-
```

The new clause is given the number 3 and is added to the current program.

• `d`: Adds a new definition (using one or more clauses).

```
Example: | ?- d.
         Do you want to check the existing definitions [n]?
         enter clauses (use the same new predicate for the heads):
         first clause: s :- p(a).
         Another definition [n]? y
         clause: s :- r, p(b), p(c).
         Another definition [n]? y
         clause: s :- r, p(b), p(a).
         Another definition [n]?
         4. s :- p(a).
         5. s :- r, p(b), p(c).
         6. s :- r, p(b), p(a).
         | ?-
```

The new clauses are clauses 4, 5, and 6, and they are added to the current program.

Unfolding rules

• `u(C,A)`: Unfolding (*) of clause `C` wrt atom `A` using the initial program. `C` is a clause number in the current program. `A` is an atom number in the body of clause C.
• `u(C)`: Short for `u(C,1)`.

• `ud_i(Pi,C,A)`: Unfolding (**) of clause `C` wrt atom `A` using program `Pi`. `Pi` is a program number, ancestor of the current program. `C` is a clause number in the current program. `A` is an atom number in the body of clause `C`.
• ud_i(Pi,C): Short for `ud_i(Pi,C,1)` using program `Pi`.

• `ud(C,A)`: Unfolding (**) of clause `C` wrt atom `A` using the initial program. `C` is a clause number in the current program. `A` is an atom number in the body of clause `C`.
• `ud(C)`: Short for `ud(C,1)`.

• `ud_p(C,A)`: Unfolding (**) of clause `C` wrt atom `A` using the current program. `C` is a clause number in the current program. `A` is an atom number in the body of clause C.
• `ud_p(C)`: Short for `ud_p(C,1)`.

• `ud_facts(C)`: Unfolding (**) of clause `C` wrt all atoms which are instances of facts.
• `ud_facts`: Unfolding (**) of all clauses of the current program wrt all atoms which are instances of facts.

(*) The unfolding wrt an atom which does not unify with the head of any clause fails (clause deletion is not performed).

(**) The unfolding of `C` wrt an atom whose predicate never occurs in the head of any clause fails. Otherwise, if the unfolding of `C` wrt an atom whose predicate occurs in the head of a clause and that atom does not unify with the head of any clause, then clause `C` is deleted.

• `ud_eq(C)`: Unfolding of clause `C` wrt all equalities.
• `ud_eq`: Unfolding of all clauses of the current program wrt all equalities.

```
Example: Assume that the current program is the following:
         1. p(A) :- q(A,B), r(B).
         2. q(a,b).
         3. q(b,a).
         4. r(a) :- q(A,B), q(B,A).
         5. r(b) :- q(A,A).
         6. r(c).
         | ?- u(1,2).
         by unfolding clause 1 w.r.t. atom 2 we get:
         7. p(A) :- q(A,a).
         8. p(A) :- q(A,b), q(B,B).
         9. p(A) :- q(A,c).
         | ?- u(8,2).
         no.
         | ?- ud(8,2).
         deletion of clause 8 because of failure in body
         | ?- u(4).
         by unfolding clause 4 w.r.t. atom 1 we get:
         10. r(a) :- q(b,a).
         11. r(a) :- q(a,b).
         | ?- ud_facts.
         deletion of clause 9
         by unfolding clause 10 w.r.t. atom(s) [1] we get:
         12. r(a).
         by unfolding clause 11 w.r.t. atom(s) [1] we get:
         13. r(a).
         deletion of clause 5
         | ?-
```

Folding rules

• `ts_f(C,As,D)`: Tamaki&Sato folding of clause `C` wrt atoms `As` using definition `D`. `C` is a clause number in the current program. `As` is a list of atom numbers in the body of clause `C`. `D` is the number of a clause introduced by definition.

```
Example: Assume that clause 5 has been introduced by definition:
     5. p(A,B,C,D,E,F) :- s_p(A,B,E,G), s_p(C,D,E,H), plus(G,H,F).
     and clause 13 is a clause in the current program:
     13. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- s_p(B,D,I,K), mult(A,C,L),
        plus(L,M,J), plus(K,N,O), plus(O,P,M), s_p(F,H,I,N), mult(E,G,P).
     | ?- ts_f(13, [1,6,4], 5).
     by folding clause 13 w.r.t. atoms [1,6,4] using 5 we get:
     14. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- p(B,D,F,H,I,K), mult(A,C,L),
        plus(L,M,J), plus(K,N,M), mult(E,G,N).
     | ?-
```

• `f(Cls11,Cls2)`: Folding of clauses `Cls1` using clauses `Cls2`. `Cls1` is a list of clause numbers in the current program. `Cls2` is a list of clause numbers in the current program, or in an ancestor of the current program. `Cls1` and `Cls2` must be ordered so that the i-th clause in `Cls1` is folded using the i-th clause in `Cls2`. No rearrangement of atoms in the bodies of the clauses is performed. See also command `r_f`.

```
Example: Assume that the current program is made out of the following clauses
     1. p([a|A]) :- t(A), u(A), v(A).
     2. p([b|A]) :- t(A), u(A), s(A).
     3. p2(a,A) :- v(A).
     4. p2(b,A) :- s(A).
     | ?- f([1,2], [3,4]).
     by folding clauses [1,2] using clauses [3,4] we get:
     5. p([A|B]) :- t(B), u(B), p2(A,B).
     | ?-
```

• `r_f(Cls11,Cls2)`: Folding of clauses `Cls1` using clauses `Cls2`. `Cls1` is a list of clause numbers in the current program. `Cls2` is a list of clause numbers in the current program, or in an ancestor of the current program.

```
Example: Assume that the current program is made out of the following clauses
     1. p([a|A]) :- t(A), u(A), v(A).
     2. p([b|A]) :- u(A), s(A), t(A).
     3. p2(a,A) :- v(A).
     4. p2(b,A) :- s(A).
     | ?- f([1,2], [3,4]).
     by folding clauses [1,2] using clauses [3,4] we get:
     p([A|B]) :- p2(A,B), t(B), u(B).
     | ?-
```

Goal replacement rules

• `rep(C,As,L)`: Replaces the goal made out of atoms `As` in the body of clause `C` by another goal according to replacement law `L`. `C` is a clause number in the current program. `As` is a list of atom numbers in the body of clause `C`. `L` is a law number in the current theory.

```
Example: Assume that the current program contains clause 10:
         10. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- s_p(B,D,I,K), mult(A,C,L),
            plus(L,K,M), s_p(F,H,I,N), mult(E,G,O), plus(O,N,P), plus(M,P,J).
         and the current theory contains law 1:
         1. plus(A,B,C), plus(C,D,E) -> plus(A,F,E), plus(B,D,F).
         | ?- rep(10,[3,7],1).
         by replacing goal [plus(A,B,C),plus(C,D,E)] by [plus(A,F,E),plus(B,D,F)]
```

```
        in clause 10 using law 1 we get:
        11. p([A|B],[C|D],[E|F],[G|H],s(I),J) :- s_p(B,D,I,K), mult(A,C,L),
           plus(L,M,J), plus(K,N,M), s_p(F,H,I,O), mult(E,G,P), plus(P,O,N).
        | ?-
```

• `mv(C,P1,P2)`: Moves atom from position `P1` to position `P2` in the body of clause `C`.

```
Example: Assume that the current program contains clause 6:
         6. p(A) :- \+A=a, q(A,B), r(B).
         | ?- mv(6,1,3).
         by moving atom 1 to position 3 in clause 6 we get:
         8. p(A) :- q(A,B), r(B), \+A=a.
         | ?-
```

• **`add_g(C)`**: Adds a goal in the body of clause `C`. The goal to be added is entered by the user interactively.

```
Example: Assume that the current program contains clause 11:
         11. p(A) :- q(A,B), r(B).
         | ?- add_g(11).
         Goal: s(a,b).
         position of new goal [3]: 2
         by adding goal s(a,b). in clause 11 at position 2 we get:
         12. p(A) :- q(A,B), s(a,b), r(B).
         | ?-
```

• **`del_g(C,As)`**: Deletes the goal made out of atoms `As` in the body of clause `C`.

```
Example: Assume that the current program contains clause 10:
         10. p(A) :- +A=b, q(A,B), r(B), +A=a.
         | ?- del_g(10,[1,4]).
         by deleting goal [1,4] in clause 10 we get:
         11. p(A) :- q(A,B), r(B).
         | ?-
```

Clause replacement rules

• **`restr(P/A)`**: Deletes clauses from current program that are used to define predicate `P` of arity `A`.

```
Example: Assume that current program is made out of the following clauses:
         1. average(A,B) :- sumlist(A,C), len(A,D), div(C,D,B).
         2. sumlist([],0).
         3. sumlist([A|B],C) :- sumlist(B,D), plus(A,D,C).
         4. len([],0).
         5. len([A|B],C) :- len(B,D), plus(D,1,C).
         6. div(A,B,C) :- C is A/B.
         7. plus(A,B,C) :- C is A+B.
         | ?- restr(sumlist/2).
         elimination of transient clause(s) [3,4,5,7]
         | ?-
```

• **`del_cl(Cs)`**: Deletes clause(s) `Cs`. `Cs` might be a clause number or a list of clause numbers.

• **`del_f_cl(Cs)`**: Deletes among clause(s) `Cs`, those clauses which have a failing body, that is, which contain an atom whose predicate is defined in the current program but which does not unify with the head of any clause. `Cs` may be a clause number or a list of clause numbers.
• **`del_f_cl`**: short for **`del_f_cl(Cs)`** where `Cs` is the list of all clauses in the current program.

```
Example: Assume that the current program is made out of the following clauses
         2. q(a,b).
         3. r(b).
         9. p(b) :- q(b,A), r(A), +b=a.
         | ?- del_f_cl.
         deletion of clause 9 because of failure in body
         | ?-
```

• `add_cl`: Adds a clause to the current program. The new clause is typed by the user.

```
Example: Assume that the current program is made out of the following clauses
         1. p([a|A]) :- t(A), u(A), v(A).
         2. p([b|A]) :- t(A), u(A), s(A).
         3. p2(a,A) :- v(A).
         4. p2(b,A) :- s(A).
         | ?- add_cl.
         clause to be added: p([a]) :- q(a,A).
         Before which clause ? [last]: 3
         Addition of clause 5:
         1. p([a|A]) :- t(A), u(A), v(A).
         2. p([b|A]) :- t(A), u(A), s(A).
         5. p([a]) :- q(a,A).
         3. p2(a,A) :- v(A).
         4. p2(b,A) :- s(A).
         | ?-
```

• `rep_cl(C)`: Replaces clause `C` by another clause in the current program. The new clause is entered by the user interactively and it takes the place of clause `C` in the current program.
• `mv_cl(C)`: Moves clause `C` in the current program to a position which is entered by the user interactively.

• `subs(Cs)`: Applies subsumption to clauses `Cs`.
• `subs`: is a short for `subs(Cs)` where `Cs` is the list of all clauses in the current program.

```
Example: Assume that the current program is made out of the following clauses
         5. p(a) :- q(a,A), r(A).
         7. p(a) :- q(a,A).
         6. p(A) :- +A=a, q(A,B), r(B).
         | ?- subs.
         clause 5 subsumed by 7
         | ?-
```

• `split(C,A)`: Splits clause `C` w.r.t. atom `A`.

```
Example: Assume that the current program contains clause 11:
         4. p(A) :- q(A,B), r(B).
         | ?- split(4, p(a)).
         by case splitting clause 4 we get:
         5. p(a) :- q(a,A), r(A).
         6. p(A) :- +A=a, q(A,B), r(B).
         | ?-
```

Commands for starting or backtracking in the transformation tree

• `st`: Initialization of a transformation process by selecting an initial program available in the directory `Progs/`.
• `st(P)`: Initialization of a transformation process with `P` as initial program. `P` must be an existing file in the directory `Progs/`.
• `load_s`: Restores a transformation tree to be selected via a menu which lists all files of directory `Sessions/`. These files are the ones automatically produced by the `s_der` command.
• `set(P)`: Moves to program `P`, that is, `P` becomes the new current program.

Commands for selecting or editing replacement laws

• `add_l`: Adds a law to the current theory. The law is given interactively by the user.
• `del_l`: Deletes a law from the current theory. The law is selected via a menu which lists all laws of the current theory.
• `sel_l`: Adds one or more laws from a theory `T` to the current theory. `T` is selected via a menu which lists all files of directory `Theories/`.

Commands for showing information

• `p1`: Shows the initial program.
• `p`: Shows the current program. See also command `pp`.
• `p(Ps)`: Shows program(s) `Ps`. `Ps` may be a program number, or a list of program numbers. See also command `pp(P)`.
• `pp`: Shows the current program. Only clauses that are not in the initial program are shown.
• `pp(Ps)`: Shows program(s) `Ps`. `Ps` may be a program number, or a list of program numbers. Only clauses that are not in the initial program are shown.
• `h(P)`: Shows history of program `P`.
• `h`: Shows history of the current program.
• `cls(Cs)`: Shows clause(s) `Cs`. `Cs` may be a clause number or a list of clause numbers.
• `defs`: Shows the definitions introduced so far.
• `l`: Shows the laws in the current theory.
• `dir_p`: Shows the files contained in the directory `Progs/`.
• `dir_t`: Shows the files contained in the directory `Theories/`.
• `dir_s`: Shows the files contained in the directory `Sessions/`.
• `dir_f`: Shows the files contained in the directory `Final_Progs/`.

Commands for saving information

• `s_p(P)`: Saves program `P` in the directory `Final_Progs/`.
• `s_p`: Saves the current program in the directory `Final_Progs/`.
• `s_h(P)`: Saves the history of program P in the directory `Histories/`.
• `s_h`: Saves the history of the current program in the directory `Histories/`.
• `s_p(P)`: Saves program `P` in the directory `Final_Progs/`.
• `s_p`: Saves the current program in the directory `Final_Progs/`.
• `s_der`: Saves the current session in the directory `Sessions/`.

Commands for adding comments

• `com_h`: Adds comments to the current history. Running this command prompts the user to add some comments. These comments will be integrated into the current history, that is, they will appear in the text file produced by command `s_h`.

Commands for getting help

• `helps`: Shows a general help menu, that is the following lines:

```
st.            < start a new transformation process >
r.             < show available rules >
s.             < show available strategies >
c.             < show available other commands >
helps.         < show this five lines menu >
```

• `r`: Shows a menu of available transformation rules. See also command `c` and command `s`.
• `c`: Shows a menu of available commands. See also command `r` and command `s`.
• `s`: Shows a menu of available strategies. See also command `r` and command `c`.
• `man(X)`: Shows help about the use of rule, command, or strategy `X`.

```
Example: | ?- man(st).
         st:
            Initialization of a transformation process by selecting
            an initial program available in the directory Progs/
         st(P):
            Initialization of a transformation process with P
            as initial program.
            P must be an existing file in the directory Progs/.
         | ?-
```

Strategies

• `uve`: This command is an implementation of the Unnecessary Variables Elimination strategy. This strategy is defined in: Proietti, M. and Pettorossi, A.: Unfolding-Definition-Folding, in this order, for Avoiding Unnecessary Variables in Logic Programs. Theoretical Computer Science 142(1):89-124, 1995.

• `lap`: This command is an implementation of the Loop Absorbtion strategy. This strategy is defined in: Proietti, M. and Pettorossi, A.: The Loop Absorption and the Generalization Strategies for the Development of Logic Programs and Partial Deduction. Journal of Logic Programming, 16(1,2):123-161, 1993.

• `determ`: This command is an implementation of a strategy for reducing nondeterminism. This strategy is defined in: Pettorossi, A., Proietti, M., Renault, S.: Reducing Nondeterminism while Specializing Logic Programs 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 15-17, 1997, p:414-427.

C Code of the Transformer

In this appendix, we give the SICStus Prolog code of the transformer, except for the file `man.pl` which contains the definition of the command `man`, that is, the messages associated with every top-level command (see Appendix B). It includes the following files:

C.1 `top_level.pl` which consists of top level operations, that is, commands that can be issued by the user at top level. They correspond to the application of the transformation rules or other commands described in Section 6.2.

C.2 `definition_rules.pl` which consists of predicates which realize the application of the introduction definition rules.

C.3 `unfolding_rules.pl` which consists of predicates which realize the application of the unfolding rules.

C.4 `folding_rules.pl` which consists of predicates which realize the application of the folding rules.

C.5 `goal_replacement_rules.pl` which consists of predicates which realize the application of the goal replacement rules.

C.6 `clause_replacement_rules.pl` which consists of predicates which realize the application of the clause replacement rules.

C.7 `input_output.pl` which contains read and write operations on files.

C.8 `msg_user.pl` which contains messages to the users, such as: menus, error messages, help messages and general informations.

C.9 `struct.pl` which contains data structures definitions and operations on them.

C.10 `basic_op.pl` which contains various basic operations.

C.11 `environment.pl` which contains assert and retract operations for updating the transformation tree.

C.12 `uve.pl` which contains predicates for the elimination of unnecessary variable strategy.

C.13 `lap.pl` which contains predicates for the loop absorption strategy. Actually, it is made out of various files which have been implemented by T. Morgante and whose corresponding code is also given in his thesis.

C.14 `determ_prog.pl` which contains predicates for a strategy for making programs semi-deterministic as explained in Section 7.

C.15 `case_split_for_determ.pl` which contains the case split rules with predicate modes used in the `determ` strategy.

C.16 `determ_prog_util.pl` which contains basic operations used in `determ_prog.pl`.

C.17 `determ_start_parameters.pl` which contains some parameters used by the strategy `determ`. This file must be updated by the user before each application of the strategy. We give two versions of this file which are used in the two examples of Section 7.3.1 and Section 7.3.3, the first version of which is commented to allow the user to modify easily the parameters for other examples.

The header of each file consists of the name of the file and a brief description of the predicates contained in it. In case it uses some predicates that are imported from the SICStus libraries, it will also contain a directive of the form '`:- use_module(library(Lib), [Preds])`', which means that the list `[Preds]` of predicates are imported from the SICStus library `Lib`. Actually, in the whole code of the transformer, we import the following predicates:

from library `system`:

- `directory_files/2`
- `file_exists/1`
- `shell/1`

from library `lists`:

- `append/3`
- `memberchk/2`
- `member/2`
- `permutation/2`
- `nth/3`
- `nth/4`
- `min_list/2`
- `same_length/2`
- `reverse/2`

from library `terms`:

- `variant/2`
- `unify_with_occurs_check/2`
- `subsumes/2`
- `subsumes_chk/2`

C.1 top_level.pl

```
% top_level.pl
%
% This file contains the top level predicates of the transformer, i.e. the
% predicates which correspond to commands that can be issued by the user.
%
% "rules", "commands", and "strategies" are top level commands which show
% to the user the menu of all other top level commands, respectively concerned
% with transformation rules, general facilities commands and strategies.
%
% "rules" shows commands related to the application of transformation rules.
% Issuing such a command has one of the following effect:
%    - initializing a transformation tree to an initial program selected by the
%      user, or restoring a transformation tree which has been saved in the past
%    - creating a new branch in the current transformation tree labelled by the
%      chosen transformation rule. The new node is labelled by the program derived
%      thereby and which becomes the new current program
%    - no effect if the chosen transformation rule does not apply to the current
%      program (for instance the conditions of application are not satisfied or
%      wrong parameters are given)
%    - setting the value of current program to a program in the transformation tree
%     (the transformation tree is unchanged, but the pointer which points to the
%     current program is moved to another program in the tree)
%    - adding or removing a replacement law. The transformation tree is unchanged
%      but the set of available laws which can be selected for applying the
%      replacement goal transformation rule is modified.
%
% "commands" shows commands which do not affect the transformation tree. They are
%  not necessary to perform any transformation step. They provide facilities for
%  consulting or saving information related to transformation states.
%
% "strategies" shows commands associated with strategies, i.e. commands which apply
%  automatically some predefined sequences of transformation rules.
%  There are three available strategies:
%  * lap: Loop Absorption Strategy, which is described in:
%   Proietti, M. and Pettorossi, A.:
%   Construction of Efficient Logic Programs by Loop Absorption and Generalization.
%   Proceedings Meta'90 (M. Bruynooghe, ed.), Leuven, Belgium (1990), p. 57-81.
%  * uve:  Elimination of Unnecessary Variables, which is described in:
%  Proietti, M. and Pettorossi, A.:
%  Unfolding-Definition-Folding, in this order, for Avoiding Unnecessary Variables
%  in Logic Programs. Theoretical Computer Science 142 (1) 1995, p. 89-124.
%     * determ: Derivation of `semi-deterministic' programs via specialization,
%     as described in:
%     Pettorossi, A., Proietti, M., Renault, S.:
%     Reducing Nondeterminism while Specializing Logic Programs,
%     24th ACM Symposium on Principles of Programming Languages (POPL'97),
%     La Sorbonne, Paris, France, January 15-17, 1997, p. 414-427.
%
%     "lap" and "uve" are top-level predicates that must be used once the
%     initial program has been selected. They use a menu in order to get the
```

```
%     necessary parameters fromn the user. Unlike "lap and "uve", "determ"
%     does not get the necessary parameters by means of a menu. All necessary
%     parameters are supposed to have been set preliminary by the user in
%     the file determ_prog_parameters.pl which is at the root of the
%     TRANSFORMER directory. Parameters for this strategy include predicate
%     modes, optionnal use of the case split rule, various option for the
%     unfolding substrategy. The name of the initial program (which must be
%     defined in directory Progs/) and the specialized clause with which
%     the transformation process must start are considered as parameters. As a
%     result, "determ" does not require the preliminary selection of an
%     initial program neither a definition introduction transformation step.
%
%
% Top level commands have an on-line manual entry stored in file man.pl.

:- use_module(library(system), [ file_exists/1 ]).
:- use_module(library(lists), [ append/3,
                                member/2 ]).

/*---------
 ts_define: Introduce a new definition. The user is prompted to enter the new
 definition made of one clause only. It corresponds to the Tamaki and  Sato
 definition. The predicate in the head of the definition must be a  new predicate.
 There is a warning message if the head of the definition uses a variable which
 does not appear in its body.
----------*/
ts_define :-
    check_exists_transf_tree, ask_enter_definition(Head,Body),
    check_head_variable(Head,Body), predname_and_arity(Head,Pred_Head,Arity),
    check_new_pred(Head,Pred_Head,Arity), apply_ts_define(Head,Body).
/*---------
 gen_define: Introduce a new definition. The user is prompted to enter the new
 definition made of possibly several clauses. The predicate in the head of each
 clause must be the same for each clause and must be a new predicate.
---------*/
gen_define :-
   check_exists_transf_tree, ask_enter_first_definition_clause(D,Head,_),
   predname_and_arity(Head,Pred_Head,Arity), check_new_pred(Head,Pred_Head,Arity),
   ask_enter_next_definition_clause([D],Pred_Head,Arity,Defs), apply_gen_define(Defs).
/*----------
 unfold: Unfold a clause wrt an atom of its body using clauses in initial program.
 The clause number and the atom number are selected by the user via a menu. This
 unfolding rule fails when there is no head of clause of the initial program
 which unifies with the selected atom.
unfold(C): Unfold clause C wrt atom 1 (the leftmost in its body).
unfold(C,A): Unfold clause C wrt atom A.
-----------*/
unfold :-
   check_exists_transf_tree, current_clauses(CNs),
   ask_clause_number(CNs,Clause_Number),
   check_clause_with_non_empty_body(Clause_Number,Length),
   ask_position_to_unfold(Length,Pos), apply_unfold(Clause_Number,Pos).
```

```
unfold(Clause_Number) :-
    check_exists_transf_tree, check_input_clause_to_unfold(Clause_Number),
    apply_unfold(Clause_Number,1).   % Default atom is at position 1
unfold(Clause_Number,Atom_Number) :-
  check_exists_transf_tree, check_input_clause_to_unfold(Clause_Number),
  check_input_atom(Clause_Number,Atom_Number),
  apply_unfold(Clause_Number,Atom_Number).
/*----------
 unfold_d: Unfold a clause wrt an atom of its body using clauses in initial
 program. The clause number and the atom number are selected by the user via a
  menu. This unfolding deletes the clause to be unfolded when the selected atom
 is a failing goal. A failing goal is an atom which unifies with no head of
 clause in the current program, and which is defined in the initial program.
 This last condition is to prevent from deleting clauses containing bip's or
 predicates whose definitions have been omitted on purpose by the user because
 they are not necessary for the derivation.
 unfold_d(C): Unfold clause C wrt atom 1 (the leftmost in its body).
 unfold_d(C,A): Unfold clause C wrt atom A.
-----------*/
unfold_d :-
   check_exists_transf_tree, current_clauses(CNs),
   ask_clause_number(CNs,Clause_Number),
   check_clause_with_non_empty_body(Clause_Number,Length),
   ask_position_to_unfold(Length,Pos), apply_unfold_d(Clause_Number,Pos).
unfold_d(Clause_Number) :-
   check_exists_transf_tree, check_input_clause_to_unfold(Clause_Number),
   apply_unfold_d(Clause_Number,1). %Default atom is number 1
unfold_d(Clause_Number,Atom_Number) :-
   check_exists_transf_tree, check_input_clause_to_unfold(Clause_Number),
   check_input_atom(Clause_Number,Atom_Number),
   apply_unfold_d(Clause_Number,Atom_Number).
/*-----------
 unfold_d_p/[0,1,2]: same as unfold_d/[0,1,2] except that the clauses of the
 current program are used for unfolding instead of those in the initial program.
-------------*/
unfold_d_p :-
   check_exists_transf_tree, current_clauses(CNs),
   ask_clause_number(CNs,Clause_Number),
   check_clause_with_non_empty_body(Clause_Number,Length),
   ask_position_to_unfold(Length,Pos),
   apply_unfold_d_p(Clause_Number,Pos).
unfold_d_p(Clause_Number) :-
   check_exists_transf_tree, check_input_clause_to_unfold(Clause_Number),
   apply_unfold_d_p(Clause_Number,1). %Default atom is number 1
unfold_d_p(Clause_Number,Atom_Number) :-
  check_exists_transf_tree,check_input_clause_to_unfold(Clause_Number),
  check_input_atom(Clause_Number,Atom_Number),
  apply_unfold_d_p(Clause_Number,Atom_Number).
/*-----------
 unfold_d_i/[0,2,3]: same as unfold_d/[0,1,2] except that the clauses of any
 intermediary program may be used for unfolding.
-------------*/
```

```
unfold_d_i :-
   check_exists_transf_tree, current_prog(PN,CNs),
   ask_clause_number(CNs,Clause_Number),
   check_clause_with_non_empty_body(Clause_Number,Length),
   ask_position_to_unfold(Length,Pos),
   ancestors(PN,Ancestors), ask_program_for_unfolding(PN,Ancestors,Pi),
   apply_unfold_d_i(Pi,Clause_Number,Pos).
unfold_d_i(Program_Number,Clause_Number) :-
  check_exists_transf_tree, check_input_program(Program_Number),
  check_input_clause_to_unfold(Clause_Number),
   apply_unfold_d_i(Program_Number,Clause_Number,1).
unfold_d_i(Program_Number,Clause_Number,Atom_Number) :-
  check_exists_transf_tree, check_input_program(Program_Number),
  check_input_clause_to_unfold(Clause_Number),
  check_input_atom(Clause_Number,Atom_Number),
  apply_unfold_d_i(Program_Number,Clause_Number,Atom_Number).
/*--------
 unfold_d_eqs: Unfold equalities in the body of all clauses.
 unfold_d_eqs(C): Unfold equalities in the body of clause C.
----------*/
unfold_d_eqs :- check_exists_transf_tree, current_clauses(CNs), unfold_d_eqs_ok(CNs).
unfold_d_eqs([]) :- !.
unfold_d_eqs([C|L]) :- !, check_exists_transf_tree, unfold_d_eq(C), unfold_d_eqs(L).
unfold_d_eqs(C) :- unfold_d_eq(C).
unfold_d_eqs_ok([]).
unfold_d_eqs_ok([C|L]) :- apply_unfold_d_eq(C), unfold_d_eqs_ok(L).
unfold_d_eq(C) :- check_input_clause(C), apply_unfold_d_eq(C).
unfold_d_eq(C) :- msg_not_a_clause_in_current_prog(C).
/*---------------
 unfold_d_facts: Simplify the body of all clauses by unfolding all instances
 of facts, deleting clauses with a failing body, and simplifying equalities
 and inequalities when possible.
 unfold_d_facts(C): Simplify clauses body of clause C.
----------------*/
unfold_d_facts:- check_exists_transf_tree, current_clauses(L), unfold_d_facts_ok(L).
unfold_d_facts([]) :- !.
unfold_d_facts([C|L]) :-!,
  check_exists_transf_tree, unfold_d_facts(C), unfold_d_facts(L).
unfold_d_facts(C) :-!, unfold_d_facts([C]).
unfold_d_facts_ok([]).
unfold_d_facts_ok([C|L]) :- apply_simpl_body(C), unfold_d_facts_ok(L).
/*-----------
 ts_fold: Fold a clause using a definition (Tamaki and Sato's folding rule).
 The clause used to fold must have been introduced by the definition introduc-
 tion rule introducing one clause only, and in an ancestor of current program.
-------------*/
ts_fold :-
    check_exists_transf_tree, available_definitions_for_ts_fold(L),
    (L=[] -> msg_definition_clauses_empty
    ; current_clauses(CNs), ask_clause_number(CNs,C_N),
    check_clause_with_non_empty_body(C_N,Length), ts_fold_(L,Length,C_N)).
```

```
ts_fold(C,Atoms,Def) :-
    check_exists_transf_tree, check_input_clause_to_fold(C,Length),
    check_input_atoms(Length,Atoms), one_clause_definition(Def),
    apply_ts_fold(C,Atoms,Def).
ts_fold_(L,Length,C_N) :-
    ask_positions_in_body(Length,Atom_Numbers), ask_definition_number(L,Ans),
    apply_ts_fold(C_N,Atom_Numbers,Ans).
/*---------
 fold(C_Ns,C_Ns2): Fold clauses in L1 using clauses in L2. No permutations in
 clauses bodies. L2 must be clauses in an ancestor of current program.
----------*/
fold(C_Ns,C_Ns2) :-
    check_exists_transf_tree,
    check_input_clauses_for_folding(C_Ns,C_Ns2,GenH,A),
    apply_fold(C_Ns,C_Ns2,GenH,A).
/*---------
 rearrange_and_fold(L1,L2): Fold clauses in L1 using clauses in L2.
 Same as fold but permutations of atoms in clauses bodies are done if necessary.
----------*/
rearrange_and_fold([C1|CsN],[C2|CsN2]) :-  !,
    check_exists_transf_tree,
    check_input_clauses_for_folding([C1|CsN],[C2|CsN2],GenH,A),
    apply_general_fold([C1|CsN],[C2|CsN2],GenH,A).
rearrange_and_fold(C1,C2) :- rearrange_and_fold([C1],[C2]).
/*---------
  replace_goal: Replace a goal in a clause by another one using a law.
-----------*/
replace_goal :-
   check_exists_transf_tree, available_laws_for_rep(L),
   (L=[] -> msg_laws_empty
   ; current_clauses(CNs), ask_clause_number(CNs,Clause_Number),
   check_clause_with_non_empty_body(Clause_Number,Length),
   replace_goal_(L,Length,Clause_Number)).
replace_goal_(L,Length,Clause_Number) :-
   ask_positions_in_body(Length,Atom_Numbers), ask_law_number(L,Ans),
   apply_replace_goal(Clause_Number,Atom_Numbers,Ans).
replace_goal(C,Atoms,Law) :-
   check_exists_transf_tree, check_input_clause_for_goal_rep(C,Length),
   check_input_atoms(Length,Atoms), is_available_law(Law),
   apply_replace_goal(C,Atoms,Law).
/*-----------
 move_lit: Move an atom within the body of a clause. The initial and final
 positions are selected by the user via a menu.
 move_lit(C,P1,P2): Move atom from position P1 to position P2 in the body of C.
-------------*/
move_lit :-
   check_exists_transf_tree, current_clauses(CNs),
   ask_clause_number(CNs,Clause_Number),
   check_clause_with_non_empty_body(Clause_Number,Length),
   ask_initial_position(Length,Pos1), ask_final_position(Length,Pos2),
   apply_move_lit(Clause_Number,Pos1,Pos2).
```

```
move_lit(Clause_Number,Pos1,Pos2) :-
   check_exists_transf_tree,
   check_input_clause_for_move_lit(Clause_Number,Length),
   check_input_atoms(Length,[Pos1,Pos2]),
   apply_move_lit(Clause_Number,Pos1,Pos2).
/*-----------
 add_goal: Add a goal in the body of a clause. The clause and the position of
 the new goal are selected by the user.
-----------*/
add_goal :-
   check_exists_transf_tree, current_prog(P,CNs), show_program(P),
   ask_clause_number(CNs,Clause_Number),ask_enter_goal_to_add(L),
   length_body_clause(Clause_Number,Length_Body),
   ask_position_of_new_goal(Length_Body,Pos),
   apply_add_goal(Clause_Number,Pos,L).
add_goal(Clause_Number) :-
   check_exists_transf_tree, check_input_clause(Clause_Number),
   ask_enter_goal_to_add(L), length_body_clause(Clause_Number,Length_Body),
   ask_position_of_new_goal(Length_Body,Pos),
   apply_add_goal(Clause_Number,Pos,L).
/*-------------
 delete_goal: Delete a goal in the body of a clause. The clause and the
 position of the goal to delete are selected by the user.
-------------*/
delete_goal :-
   check_exists_transf_tree, current_clauses(CNs),
   ask_clause_number(CNs,Clause_Number),
   check_clause_with_non_empty_body(Clause_Number,Length),
   ask_positions_in_body(Length,Atom_Numbers),
   apply_delete_goal(Clause_Number,Atom_Numbers).
delete_goal(Clause_Number,Atom_Numbers) :-
   check_exists_transf_tree,
   check_input_clause_for_goal_deletion(Clause_Number,Length),
   check_input_atoms(Length,Atom_Numbers),
   apply_delete_goal(Clause_Number,Atom_Numbers).
/*------------
 restriction(P/A): Eliminates transient definitions, that is, clauses that
 define a predicate that is never use in the body of a clause in the program
 which defines predicate P of arity A.
--------------*/
restriction(P/A) :-
  check_exists_transf_tree, check_exists_predicate(P/A), current_clauses(Cls),
  apply_restriction([P/A], Cls, Cls2),
  (seteq(Cls,Cls2) -> msg_unchanged_after_elimination(P/A) ; true).
/*---------
 delete_clauses: Delete clauses whose number are given by the user via a menu.
 delete_clauses(L): Delete clauses in L.
-----------*/
delete_clauses :- check_exists_transf_tree, current_clauses(CNs),
   ask_clause_number(CNs,Clause_Number), apply_delete_clause(Clause_Number),
   (yesno('Another clause to delete', n) -> delete_clauses ; true).
delete_clauses([]).
```

```
delete_clauses([C|Cs]) :- delete_clause(C), delete_clauses(Cs).
delete_clause(Clause_Number) :-
    check_exists_transf_tree, check_input_clause(Clause_Number),
    apply_delete_clause(Clause_Number).
delete_clause(Clause_N) :-  msg_not_a_clause_in_current_prog(Clause_N).
/*-----------
 delete_failing_body_clauses: Delete all clauses with a failing body.
 delete_failing_body_clauses(L): Delete clauses in L with a failing body.
 A failing body is a body which contains a failing goal.
-------------*/
delete_failing_body_clauses :-
   check_exists_transf_tree, current_clauses(CNS),
   delete_failing_body_clauses(CNS).
delete_failing_body_clauses(CNS) :-
   check_exists_transf_tree, delete_failing_body_clauses(CNS,CNS,Applied),
   (Applied==applied -> true ;  msg_unchanged_after_failing_bodies).
delete_failing_body_clauses([],_,_) :- !.
delete_failing_body_clauses([C|L],CNs,Applied) :-!,
   member(C,CNs), !, body_of_clause(C,R), (non_failed_body(R,CNs) -> true
   ; (Applied=applied, apply_del_cl_failing_bd(C))),
   delete_failing_body_clauses(L,CNs,Applied).
delete_failing_body_clauses([C|L],CNS,App) :-
   msg_not_a_clause_in_current_prog(C), delete_failing_body_clauses(L,CNS,App).
delete_failing_body_clauses(C,A,B) :- delete_failing_body_clauses([C],A,B).
/*---------
 add_clause: Add a clause to the current program at a position which is given
 by the user.
-----------*/
add_clause :-
    check_exists_transf_tree, current_clauses(CNs),ask_enter_clause_to_add(C),
    ask_position_introduction(CNs,Pos), apply_add_clause(C,Pos).
/*---------
 replace_clause(C): Replace clause C by another clause given by the user.
-----------*/
replace_clause(CN) :- check_exists_transf_tree, check_input_clause(CN),
    ask_enter_replacing_clause(C), apply_replace_clause(CN,C).
/*---------
 move_clause(C): Replace clause C by another one. The new clause is given by
 the user.
-----------*/
move_clause(CN) :-
    check_exists_transf_tree, check_input_clause(CN),
    current_clauses(CNs), ask_position_introduction(CNs,Pos),
    apply_move_clause(CN,Pos).
/*-----------
 subsumption: Apply subsumption to current program.
 subsumption(Cs): Apply subsumption to clauses in Cs.
-------------*/
subsumption :-
    check_exists_transf_tree, current_clauses(CNs),
    apply_subsumption_clauses(CNs,CNs2),
    (CNs=CNs2 -> msg_unchanged_after_subsumption ; true).
```

```
subsumption(Cs) :-
   check_exists_transf_tree,
   (check_input_clauses(Cs)->
            (apply_subsumption_clauses(Cs,Cs2),
             (Cs==Cs2 -> msg_unchanged_after_subsumption ; true)) ; true).
/*-----------
 split_clause/2: Case split without predicates modes
-------------*/
split_clause(Clause_Number,Atom) :-
     check_exists_transf_tree, check_input_clause(Clause_Number),
     apply_split_clause(Clause_Number,Atom,Clauses, Applied),
     (Applied== applied -> ( add_clauses(NewCNs,Clauses,nil),
     update_programs((Clause_Number, case_split, NewCNs), _),
     new_clauses(case_split,Clause_Number,NewCNs)) ; true).
/*-----------
 global_case_split/1: Case split by global analysis
-------------*/
global_case_split([C|Cls]) :- apply_case_split_global([C|Cls],_,_).
/*-----------
 binary_case_split/2: Case split of a clause wrt another clause
-------------*/
case_split(Clause_Number,Clause_Number2):-
   check_input_clause(Clause_Number), check_input_clause(Clause_Number2), !,
   apply_case_split(Clause_Number,Clause_Number2,Clauses),
   add_clauses(NewCNs,Clauses,nil),
   update_programs((Clause_Number,(case_split,Clause_Number2), NewCNs), _),
   new_clauses(case_split,Clause_Number,Clause_Number2,NewCNs).
/*---------------
 start: Initialize a new transformation process with an initial program to be
 selected by the user in the menu made out of the files stored in subdirectory
 Progs/
 start(P): Initialize a new transformation process with P for initial program.
 P must be the name of a file stored in subdirectory Progs/
-----------------*/
start :-
   erase_environment, (input -> init_environment; msg_failure_no_selection).
start(P) :- file_exists(P), !, apply_start(P).
start(P) :- my_concat_atom(['Progs/',P],F), file_exists(F), !, apply_start(F).
start(P) :- msg_unknown_file(P), fail.
/*---------
 load_session: Restore a transformation tree corresponding to a derivation
 which has been saved by the user in a previous session with the command s_sess.
 The user selects the desired transformation tree via a menu made out of all
 files in subdirectory Sessions/ .
-----------*/
load_session :-
   erase_environment, (input_session -> true ; msg_failure_no_selection).
/*----------------
 set_current_program(N): Set current program to program N.
------------------*/
set_current_program(NP) :-
   check_exists_prog_in_transf_tree(NP), apply_set_current_program(NP).
```

```
/*-----------------
 add_law_in_theory(T): Add a law to a file T to be selected by the user in the
 menu made out of the files stored in subsirectory Theories/. The law to add
 is given by the user. It is a term of the form A1,...,An -> B1,...,Bn.
 add_law_in_theory: add a law to the current theory.
--------------------*/
add_law_in_theory(T) :-
    my_concat_atom(['Theories/',T],F), file_exists(F), !,
    apply_add_law_in_theory(F).
add_law_in_theory(T) :- msg_unknown_file(T), fail.
add_law_in_theory :- check_exists_transf_tree, msg_show_existing_laws,
    ask_enter_law(Law), check_law(Law), apply_add_law(Law).
/*-----------------
 delete_law_from_theory(T): Delete a law from a file T to be selected by the
 user in the menu made out of the files stored in subsirectory Theories/.
delete_law_from_theory: Delete a law from current theory.
-------------------*/
delete_law_from_theory(T) :-
    my_concat_atom(['Theories/',T],F), file_exists(F), !,
    apply_delete_law_from_theory(T,F).
delete_law_from_theory(T) :- msg_unknown_file(T), fail.
delete_law_from_theory :- check_exists_transf_tree, available_laws_for_rep(L),
    (L=[] -> msg_laws_empty ; ask_law_number(L,Ans), apply_delete_law(Ans)).
/*-------------
 select_law_in_theory: Add to the current theory a law from a theory to be
 selected by the user in the menu made out of the files stored in directory
 Theories/
--------------*/
select_law_in_theory :-
    check_exists_transf_tree, my_directory_files('Theories',LL),
    non_empty_directory(LL,L), !,
    (select_law_in_theory(1,L,L) -> true ; msg_no_selection).
select_law_in_theory :- msg_no_available_theories.

/*--------
 show_program(N): Show program N.
 show_short_program(N): Show program N omitting clauses occurring in the
 initial one.
-----------*/
show_program(N) :-
  check_exists_transf_tree, label_node(N,Clause_Numbers), !,
  apply_show_program(Clause_Numbers).
show_program(N) :- msg_unknown_program(N).
show_short_program(1) :- show_program(1).
show_short_program(N) :-
  check_exists_transf_tree, label_node(N,Clause_Numbers), !,
      apply_show_short_program(Clause_Numbers).
show_short_program(N) :- msg_unknown_program(N).
```

```
/*--------
 show_programs(Ps): Show programs in list Ps.
 show_short_programs(Ps): Show programs in Ps omitting clauses occurring
 in the initial one.
-----------*/
show_programs(L) :- check_exists_transf_tree, show_programs_ok(L).
show_programs_ok([]).
show_programs_ok([N|Ns]) :-
   write('program '), write(N), write(':'), nl,
   (label_node(N,Clause_Numbers) ->
      apply_show_program(Clause_Numbers)
   ; write(' does not exist.')), nl, show_programs_ok(Ns).
show_short_programs(L) :- check_exists_transf_tree, show_short_programs_ok(L).
show_short_programs_ok([]).
show_short_programs_ok([N|Ns]) :-
   write('program '), write(N), write(':'), nl,
   (label_node(N,Clause_Numbers) ->
   (N = 1 ->  apply_show_program(Clause_Numbers)
   ; apply_show_short_program(Clause_Numbers)) ;
   write(' does not exist.')), nl, show_short_programs_ok(Ns).
/*-----------------
 show_initial_program: Shows initial program.
-------------------*/
show_initial_program :- check_exists_transf_tree, apply_show_initial_program.
/*-----------------
 show_current_program: Show current program.
 show_current_short_program: Show current program, omitting clauses occurring
 in the initial one.
-------------------*/
show_current_program :-
    check_exists_transf_tree, current_clauses(NCs),
    apply_show_current_program(NCs).
show_current_short_program :-
     check_exists_transf_tree, current_clauses(Clause_Numbers),
     apply_show_short_program(Clause_Numbers).
/*---------------
 show_history: Show history of the current program.
 show_history(N): Shows history of program N
-----------------*/
show_history :- check_exists_transf_tree, current_program(N), show_history(N).
show_history(N) :- check_exists_prog_in_transf_tree(N), apply_show_history(N).
/*----------
 show_clauses: Show clauses
------------*/
show_clauses :- check_exists_transf_tree, apply_show_clauses.
show_clauses(L) :- check_exists_transf_tree, show_clauses_ok(L).
show_clauses_ok([]) :- !.
show_clauses_ok([N|L]) :- !, show_clauses_ok(N), nl, show_clauses_ok(L).
show_clauses_ok(N) :- clause_number_to_clause(N,C), !, show_numb_clause((N,C)).
show_clauses_ok(N) :- msg_unknown_clause(N).
```

```
/*---------
 show_definitions: Show existing definitions
----------*/
show_definitions:-
   check_exists_transf_tree, check_exists_def, apply_show_definition.
show_definitions:- msg_definition_clauses_empty.
/*---------
 show_laws: Show current theory
----------*/
show_laws :- check_exists_law, apply_show_law.
show_laws :- msg_laws_empty.
/*----------
 dir_p: Show the content of subdirectory Progs/
------------*/
dir_p :- my_directory_files('Progs',LL), non_empty_directory(LL,L), !,
     show_available_programs(1,L,L,_),!.
dir_p :-  msg_no_available_progs.
/*----------
 dir_t: Show the content of subdirectory Theories/
------------*/
dir_t :- my_directory_files('Theories',LL), non_empty_directory(LL,L), !,
    show_available_theories(1,L,L,_),!.
dir_t :-  msg_no_available_theories.
/*----------
 dir_s: Show the content of subdirectory Sessions/
------------*/
dir_s :- my_directory_files('Sessions',LL), non_empty_directory(LL,L), !,
    show_available_sessions(1,L,_),!.
dir_s :-  msg_no_available_sessions.

/*----------
 dir_f: Show the content of subdirectory Final_Progs/
------------*/
dir_f :- my_directory_files('Final_Progs',LL), non_empty_directory(LL,L), !,
   show_available_final_programs(1,L,L,_),!.
dir_f :-  msg_no_available_final_progs.
/*-------------
 save_prog(P): Save program P. By default, this file is stored in Final_Progs/,
 unless the user specifies another name. P is the current program if omitted.
 save_program: save current program.
---------*/
save_prog(P) :-
  check_exists_transf_tree, current_program(N),
  check_exists_prog_in_transf_tree(P), set(P),
  save_current_program, set(N).
save_program :- save_current_program.
save_current_program :- check_exists_transf_tree,
    current_clauses(NCs), msg_enter_name_program(File2),
    name(Name_File2,File2), msg_save_current_program(Name_File2),
    tell(Name_File2), apply_show_only_clauses(NCs), told.
```

```
/*--------------------
 save_history(PN): Produce a file in which all programs of the transformation
 sequence from P1 to PN together with information about the rules used are
 stored. By default, this file is stored in  Histories/history_PN, unless the
 user specifies another name.
 save_current_history: save current history.
----------------------*/
save_history(N) :- check_exists_prog_in_transf_tree(N), apply_save_history(N).
save_history :-
   check_exists_transf_tree, current_program(N), apply_save_history(N).
/*-------------
 save_current_theory: Save the current theory. By default, this file is stored
 in Theories/, unless the user specifies another name.
---------*/
save_current_theory :- check_exists_law, apply_save_current_theory.
save_current_theory :- msg_laws_empty.
/*--------
 save_session : Save current transformation state
----------*/
save_session :- check_exists_transf_tree, file_input(File), name(File,L),
   append("Progs/",Prog,L), append("Sessions/Session_",Prog,History_F),
   msg_ask_save_session(History_F,File2), name(Name_File2,File2),
   msg_save_current_session(Name_File2),
   tell(Name_File2), listing_transf_tree, told.
/*---------
 insert_comment_history: Allow the user to enter some comments which will be
 associated with the current program in the history of the derivation.
----------*/
insert_comment_history :-
   check_exists_transf_tree, current_program(P),
   apply_insert_comment_history(P,[]).
/*---------
 helps: Show a general help menu
----------*/
helps :- msg_helps.
/*---------
 rules: Show the list of top-level commands related to transformation rules.
----------*/
rules :- msg_help_rules.
/*---------
 commands: Show the list of top-level commands which provide facilities for
 consulting or saving informations related to transformation states.
----------*/
commands :- msg_help_commands.
/*---------
 strategies: Show the list of top-level commands related to strategies
----------*/
strategies :- msg_help_strategies.
```

C.2 definition_rules.pl

```
% definition rules.pl
%
% This file contains the predicates which specify the effects of applying the
% transformation rules associated with the predicates ts_define and gen_define
% which are defined in top_level.pl, after selection and checking of the input
% parameters.

:- use_module(library(terms), [ variant/2 ]).

/*---------
 ts_define(H,B): Introduce a new definition with head H and body B.
----------*/
apply_ts_define(Head,Body) :-
     current_prog(PN,CNs),
     ((label_arc(PN,(def,NewCN),PNSucc),
     NewCN=[N], clause3(N,(H,B),[def|_]),
     variant((Head,Body),(H,B))) -> (label_node(PNSucc,CNsSucc),
     update_current_prog((PN,CNs),(PNSucc,CNsSucc)))
     ; (add_clauses(NewCN,[(Head,Body)],[def|_]),
     nl, show_clauses(NewCN),
     update_programs((def,NewCN),PNSucc))).

/*---------
 apply_gen_define(L): Introduce a new definition made out of clauses in L,
 where L is a list of terms of the form (Head,Body) .
----------*/
apply_gen_define(L) :- apply_gen_define(L,_).
apply_gen_define(L2,NewCN) :-
     add_clauses_gen_define(NewCN,L2), nl,
     show_clauses(NewCN),
     update_programs((gen_def,NewCN),_).

add_clauses_gen_define(L1,L2) :-
     max_clause(N), gen_X(L2,N,X), add_clauses(L1,L2, [gen_def|X]).

gen_X([],_,[]).
gen_X([_|L2],N,[N2|L3]):- N2 is N + 1, gen_X(L2,N2,L3).
```

C.3 unfolding_rules.pl

```
% unfolding_rules.pl
%
% This file contains the predicates which specify the effects of applying the
% transformation rules associated with the predicates unfold, unfold_d,
% unfold_d_i,unfold_d_facts, unfold_d_eqs, defined in top_level.pl, after
% selection and checking of the input parameters.

:- use_module(library(lists), [ append/3, member/2 ]).
:- use_module(library(terms), [ variant/2, unify_with_occurs_check/2 ]).

/*-------
 apply_unfold(C,A): Unfold clause C wrt atom A using clauses in initial program.
---------*/
apply_unfold(Clause_Number,Atom_Number) :-
   current_prog(PN,CNs),
   (label_arc(PN,(Clause_Number,(unf,Atom_Number),NewCNs),PNSucc) ->
   (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)))
   ; (apply_unfold(Clause_Number,Atom_Number,Clauses),
   add_clauses(NewCNs,Clauses,nil),
   update_programs((Clause_Number,(unf,Atom_Number),NewCNs), PNSucc))),
   new_clauses(unfold,Clause_Number,Atom_Number,NewCNs).
apply_unfold(Clause_Number,Atom_Number,Clauses) :-
   bagof((Head,Body), one_unf(Clause_Number,Atom_Number,(Head,Body)),Clauses).

one_unf(Clause_Number,_,(Head,[])) :-
   clause_number_to_clause(Clause_Number,(Head,[])), !.
one_unf(Clause_Number,Atom_Number,(Head,Body)) :-
   clause_number_to_clause(Clause_Number,(Head,Body1)),
   split(Body1,Atom_Number,Before,Atom,After),
   nonvar(Atom), Atom = (Var=Term), Var=Term, append(Before,After,Body).
one_unf(Clause_Number,Atom_Number,(Head,Body)) :-
   clause_number_to_clause(Clause_Number,(Head,Body1)),
   split(Body1,Atom_Number,Before,Atom,After), clause3(_,(Atom2,R),prog),
   unify_with_occurs_check(Atom2,Atom), append(Before,R,After,Body).
/*-------
 apply_unfold_d(C,A): Unfold_d clause C w.r.t. atom A, i.e. unfold C w.r.t. A and
 delete C if A corresponds to a failing goal.
---------*/
apply_unfold_d(Clause_N,Atom_Number) :-
   current_prog(PN,CNs),
   (label_arc(PN,(Clause_N,(uunf,Atom_Number),NewCNs),PNSucc) ->
   (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)),
   new_clauses(unfold_d,Clause_N,Atom_Number,NewCNs));
   (apply_unfold(Clause_N,Atom_Number,Clauses) ->
   (add_clauses(NewCNs,Clauses,nil),
   update_programs((Clause_N,(uunf,Atom_Number),NewCNs), PNSucc),
   new_clauses(unfold_d,Clause_N,Atom_Number,NewCNs))   ;
   (undefined_predicate(Clause_N,Atom_Number,Pred) ->
   (msg_undefined_predicate(Pred), fail) ; apply_del_cl_failing_bd(Clause_N)))).
```

```
/*-------
 apply_unfold_d_p(C,A): Unfold_d clause C w.r.t. A, using clauses in current program.
---------*/
apply_unfold_d_p(Clause_Number,Atom_Number) :- current_prog(PN,CNs),
  (label_arc(PN,(Clause_Number,(uunf_i,PN,Atom_Number),NewCNs),PNSucc) ->
  (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)),
  new_clauses(unfold_d_p,Clause_Number,Atom_Number,NewCNs));
  (apply_unfold_d_p(Clause_Number,Atom_Number,Clauses) ->
  (add_clauses(NewCNs,Clauses,nil),
  update_programs((Clause_Number,(uunf_i,PN,Atom_Number),NewCNs), PNSucc),
  new_clauses(unfold_d_p,Clause_Number,Atom_Number,NewCNs)) ;
  (undefined_predicate(Clause_Number,Atom_Number,Pred) ->
  (msg_undefined_predicate(Pred), fail) ; apply_del_cl_failing_bd(Clause_Number)))).
apply_unfold_d_p(Clause_Number,Atom_Number,Clauses) :- current_program(PN),
   bagof((Head,Body), one_unf_i(PN,Clause_Number,Atom_Number,(Head,Body)),Clauses).
/*-------
 apply_unfold_d_i(Pi,C,A): Unfold_d clause C w.r.t. atom A, using clauses in Pi.
---------*/
apply_unfold_d_i(Pj,Clause_Number,Atom_Number) :- current_prog(PN,CNs),
  (label_arc(PN,(Clause_Number,(uunf_i,Pj,Atom_Number),NewCNs),PNSucc) ->
  (label_node(PNSucc,CNsSucc),  update_current_prog((PN,CNs),(PNSucc,CNsSucc)),
  new_clauses(unfold_d_i,Pj,Clause_Number,Atom_Number,NewCNs));
  (apply_unfold_d_i(Clause_Number,Atom_Number,Pj,Clauses) ->
  (add_clauses(NewCNs,Clauses,nil),
  update_programs((Clause_Number,(uunf_i,Pj,Atom_Number),NewCNs), PNSucc),
  new_clauses(unfold_d_i,Pj,Clause_Number,Atom_Number,NewCNs)) ;
  (undefined_predicate(Clause_Number,Atom_Number,Pred) ->
  (msg_undefined_predicate(Pred), fail) ; apply_del_cl_failing_bd(Clause_Number)))).
apply_unfold_d_i(Clause_Number,Atom_Number,Pj,Clauses) :-
    bagof((Head,Body), one_unf_i(Pj,Clause_Number,Atom_Number,(Head,Body)),Clauses).
one_unf_i(_,Clause_N,_,(Head,[])) :- clause_number_to_clause(Clause_N,(Head,[])),!.
one_unf_i(Pj,Clause_Number,Atom_Number,(Head,Body)) :-
    current_program(Pj), clause_number_to_clause(Clause_Number,(Head,Body1)),
    split(Body1,Atom_Number,Before,Atom,After), nonvar(Atom), Atom=(Var = Term),
    Var=Term, append(Before,After,Body).
one_unf_i(Pj,Clause_Number,Atom_Number,(Head,Body)) :-
    clause_number_to_clause(Clause_Number,(Head,Body1)),
    split(Body1,Atom_Number,Before,Atom,After), label_node(Pj,Clauses_Pj),
    member(D,Clauses_Pj), clause_number_to_clause(D,(HeadD,BodyD)),
    unify_with_occurs_check(HeadD,Atom), append(Before,BodyD,After,Body).
/*-----------
 apply_simpl_body(C): Simplify the body of clause C by unfolding instances of facts,
 deleting clauses with a failing body, and simplifying equalities and inequalities
 if any. This predicates is used by the top level predicate unfold_d_units.
------------*/
apply_simpl_body(C) :- clause_number_to_clause(C,HB), apply_simpl_body(C,HB,_).
apply_simpl_body(C,(H,B), X):-
     apply_simpl_body(B,B2,0,[],Lit_Numbers),
     (Lit_Numbers==failing -> (apply_delete_clause(C), X=removed(C));
     (Lit_Numbers==[] -> X=C; (add_clauses(NewCN,[(H,B2)],nil), NewCN=[X],
     update_programs((C, (unfold_facts,Lit_Numbers),NewCN),_),
     new_clause(unfold_facts,C,Lit_Numbers,NewCN)))).
```

```
apply_simpl_body([\+(A=B)|_],_,_,_,failing) :- A==B, !.
apply_simpl_body([\+(A=B)|R],[\+(A=B)|R2],N,L,L2) :-
    \+ \+ unify_with_occurs_check(A,B), !, N2 is N + 1,
   apply_simpl_body(R,R2,N2,L,L2).
apply_simpl_body([\+(_=_)|R],R2,N,L,L2) :-
   !, N2 is N + 1, apply_simpl_body(R,R2,N2,[N2|L],L2).
apply_simpl_body([G|R],R2,N,L,L2) :- current_clauses(CNs), member(Clause,CNs),
  clause_number_to_clause(Clause,(H,[])), anti_unify(G,H,T), variant(T,H), !,
  N2 is N + 1, apply_simpl_body(R,R2,N2,[N2|L],L2).
apply_simpl_body([G|R],[G|R2], N,L,L2) :-
  current_clauses(CNs), member(Clause,CNs), head_of_clause(Clause, H),
  \+ \+ G = H, !, N2 is N + 1, apply_simpl_body(R,R2,N2,L,L2).
apply_simpl_body([G|R],[G|R2], N,L,L2) :-
  undefined_predicate(G), !, N2 is N + 1, apply_simpl_body(R,R2,N2,L,L2).
apply_simpl_body([G|R],[G|R2], N,L,L2) :-
  (predicate_property(G,_) ; functor(G,\+,_) ; functor(G,not,_)),!,
  N2 is N + 1, apply_simpl_body(R,R2,N2,L,L2).
apply_simpl_body([_|_],_,_,_,failing) :-!.
apply_simpl_body([],[],_,L,L).
/*--------
apply_unfold_d_eq(C): unfolds equalities and inequalities in the body of C.
----------*/
apply_unfold_d_eq(C) :- clause_number_to_clause(C, HB), apply_unfold_d_eq(C,HB,_).
apply_unfold_d_eq(C, (H,B), X) :-
  apply_unfold_d_eq(B,B2,0,[],Lit_Numbers), (Lit==failing ->
  (delete_clause(C), X=removed(C)) ; (Lit_Numbers==[] -> X=C ;
  (add_clauses(NewCN,[(H,B2)],nil), NewCN=[X],
  update_programs((C,(ud_eq,Lit),NewCN),_), new_clause(ud_eq,C,Lit,NewCN)))).
apply_unfold_d_eq(_,_,_,_,Failing) :- Failing == failing, !.
apply_unfold_d_eq([\+(A=B)|_],_,_,_,failing) :- A==B, !.
apply_unfold_d_eq([\+(A=B)|R],[\+(A=B)|R2],N,L,L2) :-
    \+ \+ unify_with_occurs_check(A,B), !, N2 is N + 1,
    apply_unfold_d_eq(R,R2,N2,L,L2).
apply_unfold_d_eq([\+(_=_)|R],R2,N,L,L2) :-
    N2 is N + 1, apply_unfold_d_eq(R,R2,N2,[N2|L],L2).
apply_unfold_d_eq([(A=B)|_],_,_,_,failing) :- \+ unify_with_occurs_check(A,B), !.
apply_unfold_d_eq([A=B|R],R2,N,L,L2) :-
   (var(A) ; var(B)), !, unify_with_occurs_check(A,B),
   N2 is N + 1, apply_unfold_d_eq(R,R2,N2,[N2|L],L2).
apply_unfold_d_eq([X=Y|R],R3,N,L,L2) :-
   X =.. [_|LX], Y =.. [_|LY], distribute_equality(LX,LY,LZ,L2), (L2 == failing -> !
   ; (N2 is N + 1, apply_unfold_d_eq(R,R2,N2,[N2|L],L2), append(LZ,R2,R3))).
apply_unfold_d_eq([G|R],[G|S], N,L,L2) :- N2 is N + 1, apply_unfold_d_eq(R,S,N2,L,L2).
apply_unfold_d_eq([],[],_,L,L).
distribute_equality([],[],[],_).
distribute_equality([X|_],[Y|_],[],failing) :- \+ X = Y, !.
distribute_equality([X|LX],[Y|LY],LZ,L2) :-
  (var(X);var(Y)), !, unify_with_occurs_check(X,Y), distribute_equality(LX,LY,LZ,L2).
distribute_equality([X|LX],[Y|LY],LZ,L2) :-
     X =.. [F|LX2], Y =.. [F|LY2], append(LX2,LX,LX3),
     append(LY2,LY,LY3), distribute_equality(LX3,LY3,LZ,L2).
distribute_equality([_|LX],[_|LY],LZ,L2) :- distribute_equality(LX,LY,LZ,L2).
```

C.4 folding_rules.pl

```
% folding_rules.pl
%
% This file contains the predicates which specify the effects of applying the
% transformation rules associated with the predicates ts_fold, fold and
% rearrange_and_fold which are defined in top_level.pl, after selection and
% checking of the input parameters.

:- use_module(library(lists), [ append/3, nth/4, min_list/2, same_length/2 ]).
:- use_module(library(terms),
                        [ variant/2, subsumes/2, unify_with_occurs_check/2 ]).
/*---------
 apply_ts_fold(C,[A1,...,An],D): Fold clause A wrt atoms at positions A1,...,An
 using clause D.
----------*/
apply_ts_fold(Clause_Number,Atom_Numbers,Def_Number) :-
  current_prog(PN,CNs),
  (label_arc(PN,(Clause_Number,(fold,Atom_Numbers,Def_Number), NewCN),PNSucc) ->
  (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)));
  (apply_ts_fold(Clause_Number,Atom_Numbers,Def_Number,Clause),
  add_clauses(NewCN,[Clause],nil),
  update_programs((Clause_Number,(fold,Atom_Numbers, Def_Number),NewCN),PNSucc))),
  new_clauses(fold,Clause_Number,Atom_Numbers,Def_Number,NewCN).
apply_ts_fold(Clause_Number,Atom_Numbers,Def_Number,Clause) :-
  (label_arc(A,(gen_def,[Def_Number])) ; label_arc(A,(def,[Def_Number]))),
  current_program(PN), (check_ancestor(A,PN) ->
  (clause_number_to_clause(Clause_Number,(H,B)),
  clause_number_to_clause(Def_Number,(H_def,B_def)),
  positions_to_elts(B,Atom_Numbers,Atoms), delete_2(Atoms,B,B1),
  ts_conditions(H_def,B_def,H,Atoms,B1), min_list(Atom_Numbers,Pos),
  nth(Pos,NewBody,H_def,B1) , Clause=(H,NewBody))
  ; msg_not_in_ancestor([Def_Number]), fail).

% ts_conditions(H_def,B_def,H,Atoms,B1): let C and D be respectively two clauses
% such that head(C)=H, body(C) is made out of the atoms of the two lists Atoms B1,
% head(D)=H_def and body(D) is made out of the atoms in list B_def.
% We check that (C,Atoms,D) satisfies the Tamaki-Sato conditions.
ts_conditions(H_def,B_Def,H,Atoms,B1) :-
  int_var(H_def,B_Def,V), subsumes(B_Def,Atoms),
  free_variables(H_def,[],[],V1), intersection_vars_empty(V,V1),
  free_variables([H|B1],[],[],V2), intersection_vars_empty(V,V2).

/*---------
 apply_fold(LC,LC2,GenH,A): Fold clauses in LC using clauses in LC2. GenH is the
 head of the folded clause, and A, the atom in its body which unifies with all
 heads of clauses in LC.
----------*/
apply_fold(Clauses_Numbers,Clauses_Numbers2,GenHead,A) :-
 (fold_cond2(Clauses_Numbers,Clauses_Numbers2,GenHead,A,NewClause) ->
 apply_fold(Clauses_Numbers,Clauses_Numbers2,NewClause)
 ; msg_impossible_folding, fail).
```

```
apply_fold(Clauses_Numbers,Clauses_Numbers2,NewClause) :-
   add_clauses(NewCN,[NewClause],nil),
   update_programs((Clauses_Numbers,(fold, Clauses_Numbers2), NewCN), _),
   new_clauses(fold,Clauses_Numbers,Clauses_Numbers2,NewCN).
/*---------
 apply_general_fold(LC,LC2,GenH,A): Fold clauses in LC using clauses in LC2.
 GenH is the head of the folded clause, and A, the atom in its body which unifies
 with all heads of clauses in LC.
 This predicate is called by the rule called rearrange_and_fold in top_level.pl.
----------*/
apply_general_fold(CsN,CsN2,GenHead,A) :-
   (general_fold_cond2(CsN,CsN2,GenHead,A,NewClause) ->
   apply_fold(CsN,CsN2,NewClause); msg_impossible_folding, fail).

% Below are defined the predicates which check the conditions for folding a list
% C1...Cn of clauses belonging to the current program wrt another list D1...Dn
% of clauses.
%
% The conditions of application are the following:
%
% 1) D1...Dn belong to a program Pj which is an ancestor of current program,
%
% 2) C1...Cn are of the form:                    % 3) D1...Dn are of the form:
%                                                %
%                                                %
% C1: (GenHead :- L, Bd(C1), R) Theta_1          %    D1:   Hd(D1) :- Bd(D1)
%               .                                %              .
%               .                                %              .
%               .                                %              .
%                                                %
% Cn: (GenHead :- L, Bd(Cn), R) Theta_n          %    Dn:   Hd(Dn) :- Bd(Dn)
%                                                %
%
% 4) D1...Dn are unifiable to A via mgu Theta_1...Theta_n
%
% 5) For any clause D: Hd(D) :- Bd(D) in Pj different from D1...Dn, Hd(D) is not
%     unifiable to A
%
% If the above conditions are satisfied, the folding of C1...Cn using
% D1...Dn is possible and the folded clause is Hd :- L, A, R.

/*------------
 check_input_clauses_for_folding(LC,LD,GenHead,A) is true if the conditions for
 folding described above are verified with LC and LD being two lists of clauses
 numbers corresponding to C1...Cn and D1...Dn respectively.
-------------*/
% failure with appropriate message if the arguments are not lists of same length.
check_input_clauses_for_folding(Clauses_Numbers,Clauses_Numbers2,_,_) :-
  \+ same_length(Clauses_Numbers,Clauses_Numbers2), !,
  msg_impossible_folding_1, fail.
```

```
% failure with appropriate message if the list of clauses wrt which the folding
% must be done do not belong to an ancestor of current program. For efficiency,
% it is first checked among the ancestors which have been derived by an intro-
% duction definition step (we expect that the clauses which are used for folding
% are more likely to be clauses that have been introduced togehter by an intro-
% duction definition step). Then it is checked systematically in all ancestors.
% If there exist an ancestor which contains D1...Dn exists, it is checked
% whether C1...Cn and D1...Dn are of the form described in the file header
% (points 2) and 3)). If it is the case, GenHead and A are returned as output
% and will be used to form the resulting folded clause. Otherwise, the folding
% step fails with an appropriate error message.
check_input_clauses_for_folding(Clauses_Numbers,Clauses_Numbers2,GenHead,A) :-
   label_arc(Anc,(gen_def,Defs)), seteq(Defs,Clauses_Numbers2), !,
   current_program(PN),
   (check_ancestor(Anc,PN) -> (heads_unifiable(Clauses_Numbers,_,GenHead),
   heads_unifiable(Clauses_Numbers2,_,A))
   ; (msg_not_in_ancestor(Clauses_Numbers2), fail)).
check_input_clauses_for_folding(Clauses_Numbers,Clauses_Numbers2,GenHead,A) :-
    current_program(PN), select_ancestor(Pj,PN), label_node(Pj,Clauses_Pj),
    subset(Clauses_Numbers2,Clauses_Pj), heads_unifiable(Clauses_Numbers2,_,A),
    heads_unifiable(Clauses_Numbers,_,GenHead),
    no_other_head_unifiable(Clauses_Numbers2,Clauses_Pj,A), !.
% if C1...Cn and D1...Dn are not of the form specified in points 2) and 3), the
% folding step is not possible. There is a failure with an appropriate message.
check_input_clauses_for_folding(_,C_N2,_,_) :- msg_impossible_folding_2(C_N2).

/*-------------
 fold_cond2(Clauses, Defs, Gen_Head, A, Folded) is true if Folded is the clause
 which corresponds to the folding of the list Clauses of clauses wrt the list
 Defs of clauses. Depending on the use of fold_cond2 (within an interactive
 folding step with parameters Clauses and Defs selected by the user or in the
 context of the strategy determ, the checking specified by predicate
 check_input_clauses_for_folding in order to fold Clauses wrt Defs has been
 performed or not. If it has been performed (interactive folding step),
 Gen_Head and A are known (non variable terms). Otherwise, they are inferred by
 using predicate "head_unifiable" (the other conditions specified in the file
 header are always satisfied when a folding step is attempted in the "determ"
 strategy. Since the presence of unit clauses create some problems, they are
 first put aside (omit_first_unit_clauses), then the folding is attempted on
 the non unit clauses (fold_cond2_), and if it succeeds, it is verified that it
 is correct also for the unit clauses (verify_recover_unit_clauses).
--------------*/
fold_cond2(Clauses, Defs, Gen_Head, A, Folded):-
   omit_first_unit_clauses(Clauses,Defs,Units,DefUnits,Clauses2,Defs2),
   fold_cond2_(Clauses2, Defs2, Gen_Head, A, Folded),
   verify_recover_unit_clauses(Folded,Units,DefUnits).
omit_first_unit_clauses([],[],[],[],[],[]).
omit_first_unit_clauses([C|Cls],[D|Defs],[C|Units],[D|DefUnits],Cls2,Defs2) :-
    is_a_unit_clause(D), !,
    omit_first_unit_clauses(Cls,Defs,Units,DefUnits,Cls2,Defs2).
omit_first_unit_clauses([C|Cls],[D|Defs],Units,DefUnits,[C|Cls2],[D|Defs2]) :-
    omit_first_unit_clauses(Cls,Defs,Units,DefUnits,Cls2,Defs2).
```

```
verify_recover_unit_clauses(_,[],[]).
verify_recover_unit_clauses(Folded,[U|Units],[D|DefUnits]) :-
    clause_number_to_clause(D, (H,[])), clause_number_to_clause(U, CU),
    copy_term(Folded,Folded2), copy_term(H,H2), unfold_body(Folded2,H2, CU),
    verify_recover_unit_clauses(Folded,Units,DefUnits).
unfold_body((H,Bs),A,U) :- member_rest(B,Bs,R), A = B, variant((H,R),U).
fold_cond2_(Cls, Defs, Gen_Head, A, Folded):-
    (var(Gen_Head) -> heads_unifiable(Cls,_,Gen_Head) ; true),
    (var(A) -> heads_unifiable(Defs,_,A) ; true),
    minimal_hd_generalization_clauses(Cls,Gen_Head,_,_,Gen_Cl),
    minimal_hd_generalization_clauses(Defs,A,_,_,Gen_Def),
    recover_folded_clause(Gen_Cl,Gen_Def, Folded).
/*---------
 minimal_hd_generalization_clauses(NumCls,Gen_Head,Cls,Gen_Cls,Dis_Cl): NumCls is
 a list of clause numbers corresponding to clauses Cls, Gen_Head is an atom which
 generalizes the head of each clause in Cls, Gen_Cls is the list Cls of clauses
 after generalization so that their heads is Gen_Head (they might contain
 equalities in their bodies) Dis_Cl is a disjunctive clause with head Gen_Head and
 whose body is the disjunction of the bodies in Gen_Cls
-----------*/
minimal_hd_generalization_clauses(NumCls,Gen_Head,Cls,Gen_Cls,Dis_Cl) :-
    Gen_Head=..[_|Args_Gen_Head],
    minimal_hd_generalization_clauses(NumCls,Args_Gen_Head,Cls,Gen_Cls),
    disjunctive_clause(Gen_Cls,Dis_Cl).
minimal_hd_generalization_clauses([],_,[],[]).
minimal_hd_generalization_clauses([N|P],Gen_Head,[(Hd,Bd)|L],[C|Cs]) :-
   clause_number_to_clause(N, (Hd,Bd)), Hd=..[F|Args],
   minimal_move_arguments_in_body(Args,Gen_Head,GenArgs,Eqs), GenHd=..[F|GenArgs],
   insere_equalities_after_inequalities(Eqs,Bd,ExtBody), C=(GenHd,ExtBody),
   minimal_hd_generalization_clauses(P,Gen_Head,L,Cs).
disjunctive_clause([(H,Bd)], (H, [Bd]) ) :-!.
disjunctive_clause([(H,Bd)|Gen_Clauses], (H, [Bd | Bds]) ) :-
    disjunctive_clause(Gen_Clauses, (H, Bds)).
minimal_move_arguments_in_body([],_,[],[]).
minimal_move_arguments_in_body([A|Args],[B|GenHead],[A|GenArgs],Eqs) :-
     var(A), occurs_at_same_pos(A,Args,B,GenHead), !, A = B,
     minimal_move_arguments_in_body(Args,GenHead,GenArgs,Eqs).
minimal_move_arguments_in_body([A|Args],[B|GenHead],[X|GenArgs],[X=A|Eqs]) :-
    var(B), !, minimal_move_arguments_in_body(Args,GenHead,GenArgs,Eqs).
minimal_move_arguments_in_body([A|Args],[B|GenHead],[X|GenArgs],Eqs) :-
    A =.. [F|ArgsA], B =.. [F|ArgsB], copy_term(B,X),
    X =.. [F|ArgsX], append(ArgsA,Args,ArgsA2),
    append(ArgsB,GenHead,GenHead2), append(ArgsX,GenArgs,GenArgs2),
    minimal_move_arguments_in_body(ArgsA2,GenHead2,GenArgs2,Eqs).
% Particular case where Args2 is a term more general than Args1
occurs_at_same_pos(V1,Args1,_,_) :- free_of_var(V1,Args1), !.
occurs_at_same_pos(_,[],_,[]).
occurs_at_same_pos(A,[T1|T1s],B,[T2|T2s]) :-
    occurs_at_same_pos_in_term(A,T1,B,T2), occurs_at_same_pos(A,T1s,B,T2s).
occurs_at_same_pos_in_term(A,T1,_,_) :- free_of_var(A,T1), !.
occurs_at_same_pos_in_term(A,T1,B,T2) :-  A == T1, B == T2.
```

```
occurs_at_same_pos_in_term(A,T1,B,T2) :-
    nonvar(T1), nonvar(T2), T1=..[_|Args1], T2=..[_|Args2],
    occurs_at_same_pos(A,Args1,B,Args2).
insere_equalities_after_inequalities(Body,[X|B2],[X|ExtBody]):-
    (var(X) ; nonvar(X), X = \+(_=_)), !,
    insere_equalities_after_inequalities(Body,B2,ExtBody).
insere_equalities_after_inequalities([],B,B) :- !.
insere_equalities_after_inequalities(Body,B2,B) :- append(Body,B2,B).
recover_folded_clause( (H1,Bds), (A,Bds2), (H1,New_Bd_C)) :-
    subseq_body_match_def( H1, Bds, A, Bds2, New_Bd_C).
subseq_body_match_def(_,[],_,[],_).
subseq_body_match_def( H1, [Bd_C|Bds], A, [Bd_D|Bds2], New_Bd_C ) :-
    free_variables(H1,[],[],VarsH1), folded_part(F,Folded_Part,G, Bd_C),
    \+ \+ unify_with_occurs_check(Folded_Part,Bd_D),
    free_variables(Bd_D,VarsH1,[],VarsInt1),
    free_variables((F,G),VarsH1,[],VarsInt2),
    free_variables(VarsInt1,VarsInt2,[],VarsInt1),
    unify_with_occurs_check(Folded_Part,Bd_D),
    free_variables(VarsInt2,[],[],Vars), VarsInt2==Vars,
    append([F,[A],G], New_Bd_C),
    subseq_body_match_def( H1, Bds, A, Bds2, New_Bd_C ).
folded_part(F, Folded_Part, G, Bd_C) :-
    body_without_equalities(Bd_C, Eqs, Real_Bd_C),
    append([F, [B1|B1s], G], Real_Bd_C), append(Eqs, [B1|B1s], Folded_Part).
/*------------
 general_fold_cond2 is a generalization of fold_cond2 where Clauses might be
 reordered to perform the folding step.
-------------*/
general_fold_cond2(Clauses, Defs, Gen_Head, A, Folded):-
    (var(Gen_Head) -> heads_unifiable(Clauses,_,Gen_Head) ; true),
    (var(A) -> heads_unifiable(Defs,_,A) ; true),
    minimal_hd_generalization_clauses(Clauses,Gen_Head,_,_,Gen_Clause),
    minimal_hd_generalization_clauses(Defs,A,_,_,Gen_Def),
    general_recover_folded_clause(Gen_Clause,Gen_Def, Folded).
general_recover_folded_clause( (H1,Bds), (A,Bds2), (H1,New_Bd_C)) :-
    subseq_body_weak_match_def( H1, Bds, A, Bds2, New_Bd_C).
% sorting is performed for efficiency reason.
subseq_body_weak_match_def(_,[],_,[],_).
subseq_body_weak_match_def( H1, [Bd_C|Bds], A, Bds2, New_Bd_C ) :-
    del_und(Bd_D,_,Bds2,Bds3), same_length(B1s,Bd_D),
    sort_list(Bd_D, Bd_D_Sorted), sort_list(Bd_C, Bd_C_Sorted),
    free_variables(H1,[],[],VarsH1), subseq(Bd_C_Sorted,B1s,FG),
    \+ \+ unify_with_occurs_check(B1s,Bd_D_Sorted),
    free_variables(Bd_D_Sorted,VarsH1,[],VarsInt1),
    free_variables(FG,VarsH1,[],VarsInt2),
    free_variables(VarsInt1,VarsInt2,[],VarsInt1),
    unify_with_occurs_check(B1s,Bd_D_Sorted),
    free_variables(VarsInt1,[],[],Vars), VarsInt1==Vars,
    [A|FG]= New_Bd_C, subseq_body_weak_match_def( H1, Bds, A, Bds3, New_Bd_C ).
```

C.5 goal_replacement_rules.pl

```
% goal_replacement_rules.pl
%
% This file contains the predicates which specify the effects of applying the
% transformation rules associated with the predicates replace_goal, move_lit,
% add_goal, and delete_goal, defined in top_level.pl, after selection and
% checking of the input parameters.

:- use_module(library(lists), [ append/3, member/2 ]).
:- use_module(library(terms), [ subsumes/2 ]).

/*-----------
 apply_replace_goal(Cl,Poss,Law): Replace goals at position Poss in clause Cl
 using law Law
------------*/
apply_replace_goal(Clause_Number,Atom_Numbers,Law_Number) :-
  current_prog(PN,CNs), member(Clause_Number,CNs),
  (label_arc(PN,(Clause_Number,(rep,Atom_Numbers, Law_Number),NewCN),PNSucc) ->
  (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)))
  ; (replacement(Clause_Number,Atom_Numbers,Atoms, Law_Number,Clause),
  add_clauses(NewCN,[Clause],aux),
  update_programs((Clause_Number,(rep,Atom_Numbers, Law_Number),NewCN), _))),
  new_clause(replace_goal,Clause_Number,Atoms,Law_Number,NewCN).

replacement(Clause_Number,Atom_Numbers,Atoms,Law_Number,(H,B)) :-
  clause_number_to_clause(Clause_Number,(H,B1)),
  (substitute(B1,Atom_Numbers,Atoms,Law_Number,B) -> true
  ; (msg_impossible_rep, fail)).

substitute(B,Atom_Numbers,Atoms,Law_Number,B1) :-
  law(Law_Number,(Left,Right)), positions_to_elts(B,Atom_Numbers,Atoms),
  delete_2(Atoms,B,B2), subsumes(Left,Atoms),
  (Atom_Numbers = [Pos|_] -> conc_position(B2,Pos,Right,B1)
  ; append(B2,Right,B1)).
/*-----------
 apply_move_lit(Cl,Pos1,Pos2):
 Move the atom of clause Cl at position Pos1 to position Pos2.
------------*/
apply_move_lit(Clause_Number,N,M) :-
  current_prog(PN,CNs), member(Clause_Number,CNs),
  (label_arc(PN,(Clause_Number,(moveatom,N,M),NewCN), PNSucc) ->
  (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)));
  (apply_move_lit(Clause_Number,N,M,Clause), add_clauses(NewCN,[Clause],nil),
  update_programs((Clause_Number,(moveatom,N,M),NewCN),PNSucc))),
  new_clause(moveatom, Clause_Number,N,M, NewCN).
apply_move_lit(Clause_Number,N,M,(H,B2)) :-
  clause_number_to_clause(Clause_Number,(H,B)), positions_to_elts(B,[N],[Atom]),
  del_1(Atom,N,B,B1), (N<M ->
  (M1 is M-1, split(B1,M1,Before,A,After), append(Before,[A|[Atom|After]],B2)) ;
  (split(B1,M,Before,A,After), append(Before,[Atom|[A|After]],B2))).
```

```
/*-----------
 apply_add_goal(Cl,Pos,L): Add goal L in clause Cl at position Pos.
------------*/
apply_add_goal(Clause_Number,Pos,L) :-
  current_clauses(CNs), member(Clause_Number,CNs),
  apply_add_goal(Clause_Number,Pos,L,Clause), add_clauses(NewCN,[Clause],nil),
  update_programs((Clause_Number,(addgoal,Pos,L),NewCN),_),
  new_clause(addgoal,Clause_Number,Pos,L,NewCN).
apply_add_goal(Clause_Number,Pos,Goal,(H,B1)) :-
  clause_number_to_clause(Clause_Number,(H,B)), Pos1 is Pos-1,
  length1(Prefix,Pos1), append(Prefix,Suffix,B), append(Prefix,Goal,Suffix,B1).
/*-----------
 apply_delete_goal(Cl,Positions): Delete atoms at position Positions in the body
 of Cl
------------*/
apply_delete_goal(Clause_Number,Atom_Numbers) :-
  current_prog(PN,CNs), member(Clause_Number,CNs),
  (label_arc(PN,(Clause_Number,(delgoal,Atom_Numbers),NewCN), PNSucc) ->
  (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)))
  ; (apply_delete_goal(Clause_Number,Atom_Numbers,Clause),
  add_clauses(NewCN,[Clause],nil),
  update_programs((Clause_Number, (delgoal,Atom_Numbers),NewCN),PNSucc))),
  new_clause(delgoal,Clause_Number,Atom_Numbers,NewCN).
apply_delete_goal(Clause_Number,Atom_Numbers,(H,B1)) :-
  clause_number_to_clause(Clause_Number,(H,B)),
  positions_to_elts(B,Atom_Numbers,Atoms), delete_2(Atoms,B,B1).
```

C.6 clause_replacement_rules.pl

```
% clause_replacement_rules.pl
%
% This file contains the predicates which specify the effects of applying the
% transformation rules associated with the predicates restriction, delete_clauses,
% delete_failing_body_clauses, add_clause, replace_clause, move_clause,
% subsumption, split_clause, which are defined in top_level.pl, after selection
% and checking of the input parameters.

:- use_module(library(terms), [ variant/2, subsumes_chk/2 ]).

/*------------
 apply_restriction(L,Cls,Cls2): Eliminates transient definitions for predicates
 in L, that is, clauses that are not used to define the predicates in L.
--------------*/
apply_restriction([P/A], Cls, Cls2) :- apply_restriction([P/A], Cls, [], Cls2).
apply_restriction(_, [], Cls, Cls) :- !.
apply_restriction(L, Cls, Inter, Cls2) :-
   member_rest(C,Cls,Cls3),  head_of_clause(C,Head),
   predname_and_arity(Head,P,A),  member(P/A,L), !,
   get_predicates_body(L,C,L2), append(Inter, [C], Inter2),
   apply_restriction(L2, Cls3, Inter2, Cls2).
apply_restriction(_, Elim, Cls, Cls) :-
    update_programs((Elim,elimclauses,[]), _), new_clause(elimclauses,Elim).

/*-----------
 apply_delete_clause(C): Delete clause C from current program.
------------*/
apply_delete_clause(Clause_Number) :-
  current_prog(PN,CNs),
  (label_arc(PN,(Clause_Number,delclause,[]), PNSucc) ->
  (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)))
  ; update_programs((Clause_Number,delclause,[]), PNSucc)),
  new_clause(delclause,Clause_Number).

/*-----------
 apply_del_cl_failing_bd(C): Delete clause C from current program. This predicate
 is used to delete a clause by application of the delete_failing_body_clauses
 rule or in the context of an unfold_d_ing step, i.e. by application of the
 unfold_d, unfold_d_p or unfold_d_i rules.
------------*/
apply_del_cl_failing_bd(C) :-
   current_prog(PN,CNs),
   (label_arc(PN,(C,delclause,[]), PNSucc) ->
   (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)))
   ; update_programs((C,delclause,[]), PNSucc)),
   new_clause(delclause_failure_body,C).
```

```
/*-----------
 apply_add_clause(C,Pos): Add clause C to current program at position Pos.
------------*/
apply_add_clause(Clause,Pos) :-
    current_prog(PN,CNs), rearrange_definition(Clause, (H,Body_list)),
    ((label_arc(PN,(Pos,addclause,NewCN),PNSucc), NewCN=[N],
    clause_number_to_clause(N,(Head,Body)),
    variant((H,Body_list),(Head,Body))) -> (label_node(PNSucc,CNsSucc),
    update_current_prog((PN,CNs),(PNSucc,CNsSucc)));
    (add_clauses(NewCN,[(H,Body_list)],nil),
    update_programs((Pos,addclause,NewCN),  PNSucc))),
    new_clause(addclause,NewCN).
/*-----------
 apply_replace_clause(C1,C2): Replace clause C1 by clause C2.
------------*/
apply_replace_clause(Cl,Clause) :-
  current_prog(PN,CNs), rearrange_definition(Clause, (H,Body_list)),
  ((label_arc(PN,(Cl,repclause,NewCN),PNSucc), NewCN=[N],
  clause_number_to_clause(N,(Head,Body)), variant((H,Body_list),(Head,Body))) ->
  (label_node(PNSucc,CNsSucc), update_current_prog((PN,CNs),(PNSucc,CNsSucc)))
  ; (add_clauses(NewCN,[(H,Body_list)],nil),
  update_programs((Cl,repclause,NewCN),  PNSucc))).
/*-----------
 apply_move_clause(C,Pos): Move clause C to position Pos
------------*/
apply_move_clause(Clause,Pos) :-
     update_programs((Pos,moveclause,Clause),_), new_clause(moveclause,Clause).
/*-----------
 apply_substitution_clauses(L1,L2): L2 is the list L1 in which the clauses which
 are subsumed by another clause in L1 are deleted.
------------*/
apply_subsumption_clauses(L1,L2):- apply_subsumption_clauses(L1,[],L2).
apply_subsumption_clauses([],L,L).
apply_subsumption_clauses([E|L],I,F) :-
   is_subsumed(E,L),!, update_programs((E,delclause,[]),_),
   apply_subsumption_clauses(L,I,F).
apply_subsumption_clauses([E|L],I,F) :-
   is_subsumed(E,I),!, update_programs((E,delclause,[]),_),
   apply_subsumption_clauses(L,I,F).
apply_subsumption_clauses([E|L],Inter,F) :-
   append(Inter,[E],Inter2), apply_subsumption_clauses(L,Inter2,F).

is_subsumed(E,[F|_]):- is_subsumed_clause(E,F),!.
is_subsumed(E,[_|L]):- is_subsumed(E,L).
% N is subsumed by M
is_subsumed_clause(N,M):-
  clause_number_to_clause(N,(H1,B1)), clause_number_to_clause(M,(H2,B2)),
  (subsumes_chk( (H2,B2), (H1,B1))
  ; (append(F,_,B1), subsumes_chk((H2,B2), (H1,F)))),
  write('clause '), write(N), write(' subsumed by '), write(M), nl.
```

```
/*-----------
 apply_split_clause(Clause_N,A,Cls,Applied): Cls is the list of the two clauses
 which results from the split of clause Clause_N wrt atom A. Case split is only
 possible if A and the head of clause Clause_N are unify. In this case Applied
 is set to constant "applied", which will indicate at some later stage that a
 transformation step has been applied. Otherwise it is left as a variable which
 will indicate that the transformation tree is unchanged.
-----------*/
apply_split_clause(Clause_N,A,Cls,Applied) :-
  clause_number_to_clause(Clause_N,(H,B)),
  (variant(H,A) -> msg_unchanged_by_case_split
  ; (\+ \+  H=A -> apply_split_clause(H,B,A,Cls,Applied)
  ; msg_unchanged_by_case_split)).
apply_split_clause(H,B,A,Cls,Applied) :-
    anti_unify(H,A, Theta_H, Theta_H2,_),
    recover_split_clause_arg(Theta_H,Theta_H2,A_H,A_H2,Applied),
    case_split_arg((H,B),A_H,A_H2,Cls).
recover_split_clause_arg([V1=A_H|_],[V1=A_H2|_],A_H,A_H2,applied):-
    var(A_H), nonvar(A_H2), !.
recover_split_clause_arg([_|L1],[_|L2],A_H,A_H2,Applied):-
    recover_split_clause_arg(L1,L2,A_H,A_H2,Applied).
recover_split_clause_arg([],[],_,_,_):- msg_unchanged_by_case_split.

case_split_arg((H,B),A_H,K, [(H,B3), (H2, [\+(A_H2=K)| B2])]):-
   copy_term((A_H,H,B),(A_H2,H2,B2)), A_H=K,
   singletons_vars_body(B,Singletons),
   simpl_equalities_body(Singletons,B,B3).
simpl_equalities_body(_,[],[]).
simpl_equalities_body(Singletons,[([A|B]=[C|D])|Bd1],[B=D|Bd2]):-
    A==C,!, simpl_equalities_body(Singletons,Bd1,Bd2).
simpl_equalities_body(Singletons,[\+([A|B]=[C|D])|Bd1],[\+(A=C)|Bd2]):-
   (belongs(D,Singletons);belongs(B,Singletons)),
   simpl_equalities_body(Singletons,Bd1,Bd2).
simpl_equalities_body(Singletons,[P|Bd1],[P|Bd2]):-
   simpl_equalities_body(Singletons,Bd1,Bd2).
singletons_vars_body(B,Sings):-
   free_variables(B,[],[],Vars), occurrences_of_vars(Vars,B,Sings).
occurrences_of_vars([V|Vars],B,[V|Sings]):-
   occurrences_of_var(V,B,1),!, occurrences_of_vars(Vars,B,Sings).
occurrences_of_vars([_|Vars],B,Sings):- occurrences_of_vars(Vars,B,Sings).
occurrences_of_vars([],_,[]).
```

C.7 input_output.pl

```
% input_output.pl
%
% This file contains the predicates which are concerned with read and
% write operations.
%
:- use_module(library(system), [ file_exists/1,
                                 shell/1 ]).
:- use_module(library(lists), [ nth/3 ]).

/*-------
 input: Shows the list of available programs in subdirectory "Progs/" and
 prompts the user to select one which becomes the initial program of a new
 transformation process. This predicate is used at top level by start.
---------*/
input  :-
  nltab, write('Available programs:'), nl, my_directory_files('Progs',LL),
  non_empty_directory(LL,L), !, show_available_programs(1,L,L,Num_P), !,
  ask_between('Initial program', 1, Num_P, 'no selection', Answer),
  (Answer = 'no selection' -> fail ; (nth(Answer,L,Prog),
  my_concat_atom(['Progs/',Prog],Path), input(Path),
  assert(file_input(Path)))).
input :- msg_no_available_progs.

input(File) :- see(File), read_list(1), seen, show_initial_program.

show_available_programs(N,[File1],L_progs,N) :- !,
    nl, write(N), write(' ) '), write(File1), nl, repeat, nl,
    (yesno('Do you want to look at a program ',n) ->
    (ask_between('Which one ?', 1, N, 'none', Answer), nl,
    (Answer = 'none' -> true ; (nth(Answer,L_progs,Prog), name(Prog,ProgAscii),
    append("more Progs/",ProgAscii, S_Unix_command),
    name(Unix_Command,S_Unix_command), shell(Unix_Command), fail ))) ; true).
show_available_programs(N,[File1|L],L_progs,NF) :-
    nl, write(N), write(' ) '), write(File1), N2 is N +1,
    show_available_programs(N2,L,L_progs,NF).

/*---------------
 select_theory(N,[T],L_theories) is used at top level by select_law_in_theory.
----------------*/
select_theory(N,[T],L_theories) :- !,
     nl, write(N), write(' ) '), write(T), nl, nl,
     ask_between('Theory number', 1, N, 'no selection', Answer),
     (Answer = 'no selection' -> fail ; (nth(Answer,L_theories,Theory),
     my_concat_atom(['Theories/',Theory], File_Theory),
     see(File_Theory), read_laws,  seen)).
select_theory(N,[T1|L],L_theories) :-
     nl, write(N), write(' ) '), write(T1), N2 is N +1,
     select_theory(N2,L,L_theories).
```

```
apply_show_initial_program :-
     nl, write('Initial program:'), nl, nl, clause3(N,C,prog),
     show_numb_clause((N,C)), nl, fail.
apply_show_initial_program.
/*-----------
 apply_show_history: Show history.
------------*/
apply_show_history(N) :-
    ancestors(N,NPs), append(NPs,[N],NPps), nl, nl, show_information(NPps).
show_information([P,P1|Ps]) :-
     (P=1 -> write_inf(1),nl; true), label_arc(P1,LA), write_inf(P,LA,P1),
     nl, show_information([P1|Ps]).
show_information([P]) :- P=1 -> write_inf(1), nl, nl;true.
/*-----------
 apply_show_definition: show available definitions (in the ancestor nodes
 of current program)
------------*/
apply_show_definition :-
   current_program(P), definition_clauses(Pr,D), check_ancestor(Pr,P),
   show_clauses_ok(D), nl, fail.
apply_show_definition.
/*-----------
apply_show_law: show available laws
------------*/
apply_show_law:- nl, law(N,L), show_law_number((N,L)), fail.
apply_show_law.
/*-----------
apply_save_history(N): save history of program N in subdirectory Histories/
in a file named hist_P_N where P is the name of the initial program for
the current transformation process.
------------*/
apply_save_history(N) :-
   file_input(File), name(File,L), append("Progs/",Name_L,L),
   append("hist_",Name_L,Hist_L), append(Hist_L,"_",L_), name(N,L2),
   append(L_,L2,ProgN), append("Histories/",ProgN,History_F),
   msg_ask_save_history(History_F,File2), name(Name_File2,File2),
   msg_save_history(L,Name_File2), ancestors(N,Ns), append(Ns,[N],Ms),
   tell(Name_File2), show_short_commented_programs(Ms), told, nl.
/*--------
show_available_final_programs used at top level by dir_f.
--------*/
show_available_final_programs(N,[File1],L_progs,N) :- !,
     nl, write(N), write(' ) '), write(File1), nl, repeat, nl,
     (yesno('Do you want to look at a program ',n) ->
     (ask_between('Which one ?', 1, N, 'none', Answer), nl,
     (Answer = 'none' -> true ; (nth(Answer,L_progs,Prog), name(Prog,Ascii),
     append("more Final_Progs/",Ascii, S_Unix),
     name(Unix,S_Unix),shell(Unix), fail ))) ; true).
show_available_final_programs(N,[File1|L],L_progs,NF) :-
     nl, write(N), write(' ) '), write(File1), N2 is N +1,
     show_available_final_programs(N2,L,L_progs,NF).
```

```
/*--------
show_available_sessions used at top level by dir_s.
--------*/
show_available_sessions(N,[File1],N) :- !,
     nl, write(N), write(' ) '), write(File1).
show_available_sessions(N,[File1|L],N3) :-
     nl, write(N), write(' ) '), write(File1), N2 is N +1,
     show_available_sessions(N2,L,N3).

/*--------
show_available_theories used at top level by dir_t.
--------*/
show_available_theories(N,[File1],L_progs,N) :- !,
   nl, write(N), write(' ) '), write(File1), nl, repeat, nl,
   (yesno('Do you want to look at a theory ',n) ->
   (ask_between('Which one ?', 1, N, 'none', Answer), nl,
   (Answer = 'none' -> true ;
   (nth(Answer,L_progs,Prog), show_theory2_ok(Prog), fail))) ; true).
show_available_theories(N,[File1|L],L_progs,NF) :-
    nl, write(N), write(' ) '), write(File1), N2 is N +1,
    show_available_theories(N2,L,L_progs,NF).
show_theory2_ok(Prog) :-
    name(Prog,ProgAscii), append("more Theories/",ProgAscii, S_Unix_command),
    name(Unix_Command,S_Unix_command), shell(Unix_Command).
select_theory(N,[T],L_theories) :- !,
    nl, write(N), write(' ) '), write(T), nl, nl,
    ask_between('Theory number', 1, N, 'no selection', Answer),
    (Answer = 'no selection' -> fail ; (nth(Answer,L_theories,Theory),
    my_concat_atom(['Theories/',Theory], File_Theory), see(File_Theory),
    read_laws,  seen)).
select_theory(N,[T1|L],L_theories) :-
    nl, write(N), write(' ) '), write(T1), N2 is N +1,
    select_theory(N2,L,L_theories).

/*-------
 input_session: Shows the list of available sessions in directory Sessions/
 and prompts the user to select the which must become the new transformation
 tree. This predicate is used at top level by load_session.
---------*/
input_session  :-
   nltab, write('Available sessions:'),nl, my_directory_files('Sessions',LL),
   non_empty_directory(LL,L), !, show_available_sessions(1,L,Num_P),!,
   ask_between('Session number', 1, Num_P, 'no selection',Ans),
   (Ans = 'no selection' -> fail
   ; (nth(Ans,L,Prog), my_concat_atom(['Sessions/',Prog],Path),
   input_session(Path))).
input_session  :- msg_no_available_sessions.
show_available_sessions(N,[File1],N) :- !,
   nl, write(N), write(' ) '), write(File1).
show_available_sessions(N,[File1|L],N3) :-
   nl, write(N), write(' ) '), write(File1), N2 is N +1,
   show_available_sessions(N2,L,N3).
```

```
input_session(File) :- see(File), read_session, seen.
read_session :-
   read(C), C\==end_of_file, !, (functor(C, library_directory, _) -> true ;
   (functor(C, file_search_path, _) -> true
   ; (functor(C, file_search_path, _) -> true ; assert(C)))), read_session.
read_session.
/*----------
 apply_add_law_in_theory(T): Add a law to file T. The law is typed by the user.
 He may add more than one law. This predicate is used at top level by
 add_law_in_theory(T).
------------*/
apply_add_law_in_theory(File_Theory) :-
   (yesno('Do you want to see the already existing laws ',n) ->
   show_theory_ok(File_Theory) ; true), nl, write('Replacement law to add: '),
   repeat, read(Law), nl,
   (check_law(Law) -> concat_law(File_Theory,Law) ; msg_syntax_law),
   (yesno('Another law to add',n) -> nl, nl, write(': '), fail ; true),
   nltab, write('New theory saved in file: '), write(File_Theory), nl .
/*----------
 apply_delete_law_from_theory(T): Delete a law from file T. The number of the
 law is chosen by the user via a menu. He may delete more than one law.
 This predicate is used at top level by delete_law_from_theory(T).
------------*/
apply_delete_law_from_theory(Theory,File_Theory) :-
   get_file(File_Theory,L), extract_file_laws(L,Laws,_),
   show_laws(0,Laws,Max), repeat, nl,
   ask_between('enter law to delete', 1, Max, 'no selection',Answer2),
   (Answer2 = 'no selection' ->  true
   ; (nth(Answer2,Laws,Law), delete_law_from_theory_file(Law,Theory))),
   (yesno('Another law to delete ',n) -> fail ; true).
show_laws(0,[],_) :- !, msg_no_law_in_theory.
show_laws(N,[],N) :- nl.
show_laws(N,[Law|Laws],Max) :-
   N2 is N+1, show_law_num(N2,Law), show_laws(N2,Laws,Max).
extract_file_laws([],[],[]).
extract_file_laws([Line|Lines], [Law|Laws],[Law2|Laws2]):-
   law_line(Law,Law2,Line),!, extract_file_laws(Lines, Laws,Laws2).
extract_file_laws([_|Lines], Laws,Laws2):-
   extract_file_laws(Lines, Laws,Laws2).

delete_law_from_theory_file(Law,File_Theory) :-
   my_concat_atom(['old_', File_Theory], Old_File),
   my_concat_atom(['mv Theories/', File_Theory], B_S_Unix_command),
   my_concat_atom([B_S_Unix_command, ' Theories/'], M_S_Unix_command),
   my_concat_atom([M_S_Unix_command, Old_File], S_Unix_command),
   shell(S_Unix_command), name(Quote, [39]),
   my_concat_atom(['sed s/', Quote, Law, Quote],X),
   my_concat_atom([X,'//g Theories/'],Y), my_concat_atom([Y, Old_File], Z),
   my_concat_atom([Z, ' > Theories/'], T), my_concat_atom([T, File_Theory], U),
   shell(U), msg_save_file_laws(File_Theory).
```

```
/*----------
select_law_in_theory(N,T,T2) is used at top level by seslect_theory.
------------*/
select_law_in_theory(_,[],_) :- !, nl,
   write('Empty file of theories (stored in sub-directory Theories/) '), nl.
select_law_in_theory(N,[T],L_theories) :- !,
   nl, write(N), write(' ) '), write(T), nl,
   ask_between('Theory number', 1, N, 'no selection', Answer),
   (Answer = 'no selection' -> fail ; (nth(Answer,L_theories,Theory),
   my_concat_atom(['Theories/',Theory], File_Theory),
   read_laws(File_Theory,LLaws), show_laws(0,LLaws,Max), nl, repeat, nl,
   ask_between('Law number', 1, Max, 'no selection', Answer2),
   (Answer2 = 'no selection' -> true ; (nth(Answer2,LLaws,Law),
   apply_add_law(Law))), (yesno('Another law to select ',n) -> fail ; true))).
select_law_in_theory(N,[T1|L],L_theories) :-
  nl, write(N), write(' ) '), write(T1), N2 is N +1,
  select_law_in_theory(N2,L,L_theories).
show_theory_ok(Prog) :-
   name(Prog,ProgAscii), append("more ",ProgAscii, S_Unix_command),
   name(Unix_Command,S_Unix_command), shell(Unix_Command).
read_laws(T,Laws) :- see(T), read_laws2([],Laws), seen.
read_laws2(L1,L) :-
   read(C), C\==end_of_file, !, (C=(_ -> _) ->
   (append(L1,[C],LL1), read_laws2(LL1,L)) ; read_laws2(L1,L)).
read_laws2(L,L).

apply_show_program(Clause_Numbers) :-
   numb_fetch_clauses(Clause_Numbers,Numbered_Clauses),
   show_numb_clauses(Numbered_Clauses).
apply_show_short_program(Cs) :-
   label_node(1,InitialClauses), subtract(Cs, InitialClauses, Clauses),
   numb_fetch_clauses(Clauses,Num_Clauses), show_numb_clauses(Num_Clauses).
apply_show_initial_program:-
    nl, write('Initial program:'), nl, nl, clause3(N,C,prog),
    show_numb_clause((N,C)), nl, fail.
apply_show_initial_program.
apply_show_current_program(NCs) :-
    numb_fetch_clauses(NCs,Cls), show_numb_clauses(Cls).

apply_show_clauses :-
    clause_number_to_clause(N,C), show_numb_clause((N,C)), nl, fail.
apply_show_clauses.
show_numb_clauses([NC|NCs]) :-
    show_numb_clause(NC), nl, show_numb_clauses(NCs).
show_numb_clauses([]).
show_numb_clause(NC) :-
    NC=(N,C), number_clause(N), !,  write(N), write('. '), show_clause(C).
show_numb_clause(C) :- show_clause(C).
show_clause((H,B)):-
    numbervars((H,B),0,_), writeq(H), (B=[] -> write('.'); write(' :- '),
    show_body(B, 10)).
```

```
/*--------------
apply_show_only_clauses is used at top level by save current program.
--------------*/
apply_show_only_clauses(NCs) :-
   numb_fetch_clauses(NCs,Cls), show_not_numb_clauses(Cls).

show_not_numb_clause(NC) :-  NC=(Num,C), number_clause(Num), !, show_clause(C).
show_not_numb_clause(C) :- show_clause(C).
show_not_numb_clauses([]).
show_not_numb_clauses([C|Clauses]) :-
   show_not_numb_clause(C), nl, show_not_numb_clauses(Clauses).

show_body(Body,PosBody) :- show_body_lines(Body,PosBody,PosBody,50).

show_body(L) :- show_body_lines(L,10,10,50).
show_body_lines([H], PosInitBody,PosLine, Max) :- !,
print_length(write(H),Length), NewPos is PosLine + Length,
(NewPos > Max -> nl, tab(PosInitBody); true),writeq(H), write('.').
show_body_lines([H|T],PosInitBody,Pos,Max) :-
print_length(write(H),Length), NewPos is Pos + Length,
(NewPos > Max -> nl, tab(PosInitBody), Pos2=PosInitBody; Pos2=NewPos),
        writeq(H), write(', '), show_body_lines(T, PosInitBody,Pos2,Max).

show_goal(L) :- show_goal_lines(L,10,10,50).
show_goal_lines([H], PosInitBody,PosLine, Max) :- !,
print_length(write(H),Length), NewPos is PosLine + Length,
(NewPos > Max -> nl, tab(PosInitBody); true), writeq(H).
show_goal_lines([H|T],PosInitBody,Pos,Max) :-
print_length(write(H),Length), NewPos is Pos + Length,
(NewPos > Max -> nl, tab(PosInitBody), Pos2=PosInitBody; Pos2=NewPos),
        writeq(H), write(', '), show_goal_lines(T, PosInitBody,Pos2,Max).

show_law_num(N,L) :- write(N), write('. '), numbervars(L,0,_), write(L), nl.
show_law_number((N,(L,R))) :-
   numbervars((N,(L,R)),0,_), write(N), write('. '), show_lhs(L),
   write(' -> '), show_rhs(R), nl.
show_lhs([]).
show_lhs([E]) :- !, write(E).
show_lhs([E|L]) :- write(E), write(', '), show_lhs(L).
show_rhs([]) :- write('.').
show_rhs([E]) :- !, write(E), write('.').
show_rhs([E|L]) :- write(E), write(', '), show_rhs(L).

apply_save_current_theory :-
   repeat, nl, write('Name of the new theory :  '),
   get_line(S_File), (S_File =[] -> msg_no_selection ;
   (my_last(S_File,Head_S_File,46), append("Theories/",Head_S_File,File2),
   name(Name_File2,File2),
   (file_exists(Name_File2) -> msg_choose_another_filename(Name_File2), fail
   ; true),
   msg_save_current_theory(Name_File2), tell(Name_File2),
   show_notnumbered_laws, told)).
```

```
show_notnumbered_laws :- nl, law(_,L), show_law(L), fail.
show_notnumbered_laws.
show_law(((L,R))) :-
   numbervars((L,R),0,_), show_lhs(L), write(' -> '), show_rhs(R), nl.
show_short_commented_programs([]).
show_short_commented_programs([N]) :-
   (N=1 -> write_inf(1), nl; true),
   (label_node(N,_) -> (show_short_program(N), nl,
   check_assert_comment_history(N); write(' does not exist.'))).
show_short_commented_programs([N,N1|Ns]) :-
   (N=1 -> write_inf(1), nl ; true),
   (label_node(N,_) -> (show_short_program(N), nl,
    check_assert_comment_history(N), label_arc(N1,LA), write_inf(N,LA,N1), nl)
    ; write(' does not exist.')), nl, show_short_commented_programs([N1|Ns]).
check_assert_comment_history(P) :-
    comment_history(P,Comment), !, write_comment_history(Comment).
check_assert_comment_history(_).

write_comment_history([]) :- nl.
write_comment_history([X|Xs]) :-
   nl, name(T,X), write(T), write_comment_history(Xs).

apply_insert_comment_history(P,L) :-
   get_line(X), (X =[] -> assert_comment_history(P,L)
   ; apply_insert_comment_history(P,[X|L])).

show_list_h([]) :- write('.'), nl.
show_list_h([E|L]) :- tab(2), write(E), show_list_h(L).
show_list_v([]) .
show_list_v([E|L]) :- nltab, write(E), show_list_v(L).
nltab :- nl, tab(5).
show_packets([P]) :- !,
   new_clauses(P,NewClauses), show_numb_clauses(NewClauses).
show_packets([P|Ps]) :-
  new_clauses(P,NewCls), show_numb_clauses(NewCls), nl, show_packets(Ps).

give_modes([],_).
give_modes([X|L],1) :-
   ask_oneof('First argument ', ['+','-'], '+', X), give_modes(L,2).
give_modes([X|L],2) :-
   ask_oneof('Second argument ', ['+','-'], '+', X), give_modes(L,3).
give_modes([X|L],3) :-
   ask_oneof('Third argument ', ['+','-'], '+', X), give_modes(L,4).
give_modes([X|L],N) :-
   ask_oneof('Next argument ', ['+','-'], '+', X), give_modes(L,N).

listing_transf_tree :-
    listing(node), listing(number_clauses), listing(clause3),
    listing(current_prog), listing(file_input), listing(max_clause),
    listing(max_law), listing(maxnode), listing(number_clauses),
    listing(law), listing(new_file_input).
```

C.8 msg_user.pl

```
% msg_user.pl
%
% This file contains some messages that may be given during a transformation
% session to the user. They might be simple comments on what the program currently
% does or more specific information to help the user answering to the program's
% requirement. Possible choices of interaction, expected types of entries, etc.

:- use_module(library(system), [ file_exists/1 ]).
:- use_module(library(lists), [ append/3 ]).

msg_help_rules :-
nl, show_list_v([
'-------------------------- UNFOLDING RULES --------------------------',
' ',
'u/[0,1,2]        < unfold (*)  using initial program >',
'ud/[0,1,2]       < unfold (**) using initial program >',
'ud_p/[0,1,2]     < unfold (**) using current program >',
'ud_i/[0,2,3]     < unfold (**) using any intermediate program >',
'ud_facts/[0,1]   < unfold (**) wrt all instances of unit clauses >',
'ud_eqs/[0,1]     < unfold (**) wrt equalities >',
' ',
'(*)   The unfolding wrt an atom which does unify with no head',
'      of clause fails, (no clause deletion).',
'(**)  The unfolding of C wrt an atom whose predicate never occurs',
'      in the head of a clause fails. Otherwise, if it does unify with ',
'      no head of clause, it produces the deletion of C.',
' ', ' ',
'--------------------------- FOLDING RULES --------------------------',
' ',
'ts_f/[0,3]       < fold one clause using a definition (Tamaki and Sato) >',
'f/2              < fold several clauses >',
'r_f/2            < fold several clauses with possible body rearragement >',
' ',' ',
'-------------------- DEFINITION INTRODUCTION RULES --------------------',
' ', ' ',
'ts_d   < introduce a new definition (Tamaki & Sato) >',
'd      < introduce a new definition (possibly several clauses) >',
' ',' ',
'---------------------- GOAL REPLACEMENT RULES ------------------------',
' ',
'rep/[0,3]        < goal replacement rule >',
'mv/[0,3]         < move of atoms in a clause body >',
'add_g/[0,1]      < add a goal in a clause body >',
'del_g/[0,1]      < delete a goal from a clause body >',
' ',' ',
'-------------------- CLAUSE REPLACEMENT RULES ------------------------',
' ',
'restr/1          < delete useless clauses w.r.t a given predicate >',
'del_cl/[0,1]     < delete clause(s) >',
'del_f_cl/[0,1]   < delete clause(s) with failing bodies >',
```

```
'add_cl          < add a clause to current program >',
'rep_cl/1        < replacement of a clause by another one >',
'subs/[0,1]      < apply subsumption >',
'split/2         < case split of a clause wrt an atom >',
' ',' ',
'-----------------------------------------------------------------------',
' ',
'Type man(rule_name) for a detailed help about rule_name.',
' ',
' ']), nl.

msg_help_commands :-
    nl, show_list_v([
'----------------- START AND NAVIGATE THE TRANSFORMATION TREE -------------',
' ',
'st/[0,1]   < start a transformation process >',
'load_s     < restore a session >',
'set/1      < move to a program in the transformation tree >',
' ',' ',
'-------------------- SELECT AND EDIT REPLACEMENT LAWS ---------------------',
' ',
'sel_l          < select a law from a theory and add it to current theory >',
'add_l          < add a law to current theory >',
'add_l/1        < add a law to a theory present in directory Theories/>',
'del_l          < delete law from current theory >',
'del_l/1        < delete law from a theory present in directory Theories/>',
' ',' ',
'---------------------------- SHOW INFORMATION ----------------------------',
' ',
'p1        < show initial program >',
'p/[0,1]   < show program(s) >',
'pp/[0,1]  < show program(s) (new clauses only) >',
'cls/1     < show clause(s) >',
'h/[0,1]   < show history of a program >',
'defs      < show definitions introduced so far>',
'l         < show laws in current theory >',
'dir_p     < show files contained in directory Progs/ >',
'dir_t     < show files contained in directory Theories/ >',
'dir_s     < show files contained in directory Sessions/ >',
'dir_f     < show files contained in directory Final_Progs/ >',
' ',' ',
'---------------------------- SAVE INFORMATION ----------------------------',
' ',
's_p        < save current program in directory Final_Progs/ >',
's_h/[0,1]  < save history of a program in directory Histories/ >',
's_th       < save current theory in directory Theories/ >',
's_sess     < save current session in directory Sessions/  >',
' ',' ',
'---------------------------- EDIT INFORMATION ----------------------------',
' ',
'com_h     < add comments to current history >',
' ',' ',
```

```
'-------------------------------- HELPS -----------------------------------',
' ',' ',
'helps     < general help menu >',
'r         < show available transformation rules >',
's         < show available strategies >',
'c         < show other commands (this menu) >',
' ',' ',
'-----------------------------------------------------------------------',
' ',
'Type man(command_name) for a detailed help about command_name.',
' ',
' ']), nl.

msg_help_strategies :-
                 nltab, write('Available strategies: '), nl,
        show_list_v([
'uve             < unnecessary variables elimination >',
'lap             < loop absorption >',
'determ          < reduction of nondeterminism via specialization >' ,
' ',
'Type man(strategy_name) for a detailed help about strategy_name.'
] ) , nl.

msg_failure_no_selection:- nltab, write('No program has been selected.'), fail.
msg_not_a_pnat :-
   write('This is not an expected entry.'), nl,
   write('Enter a natural number.'), nl.
msg_no_selection :- nltab, write('No selection.'), nl.
msg_no_action :- nltab, write('No action performed.'), nl.
msg_unchanged_after_failing_bodies :- nltab, write('Program unchanged.'), nl.
msg_unchanged_after_subsumption :-
   nltab, write('Program unchanged after subsumption.'), nl.
msg_unchanged_after_elimination(P/A) :-
   nltab, write('No transient clauses for the definition of predicate '),
   write(P/A), write('.'), nltab, write('Program unchanged.'), nl.
msg_unchanged_by_case_split :-
   nltab, write('Case split not possible, program unchanged.'), nl.
msg_unknown_predicate(Pred) :-
   nltab, write('Warning: unknown predicate '), write(Pred), nl.
msg_unknown_program(P) :-
    nltab, write('Program '), write(P), write(' does not exist.'), nl.
msg_unknown_file(P) :-
    nltab, write('Unknown file: '), write(P), nltab,
    write('Enter "add_prog" to have the list of available programs. '), nl.
msg_unknown_clause(P) :-
    nltab, write('Clause '), write(P), write(' does not exist.'), nl.
msg_not_a_clause_in_current_prog(Clause_Number):-
   write(Clause_Number), write(' is not a clause in current program'), nl.
msg_already_used_in_prog(X):-
   nltab, write(X), write(' is already used in current program.'), fail.
msg_already_used_in_def(X):-
   nltab, write(X), write(' is already used in a definition.'), fail.
```

```
msg_definition_clauses_empty :- nltab, write('No definition !'), nl.
msg_must_be_a_list_of_atoms(A) :-
   nltab, write('A list of integers is expected instead of '), write(A), nl.
msg_bad_positions_in_body :-
   nltab, write('Bad positions for selected atoms ! '), nl.
msg_impossible_folding :- nltab, write('Impossible folding.'), nl.
msg_impossible_rep:-
   nltab, write('Replacement not possible (no match), program unchanged.'), nl.
msg_impossible_folding_1 :-
   nltab, write('Impossible folding:'), nltab,
   write('Expected arguments: two lists (clauses and definitions)'),
   write(' of same length.'), nl.
msg_impossible_folding_2(Defs) :-
   nltab, write('Warning: clauses '), write(Defs),
   write(' must belong to an ancestor Pj'), nltab,
   write('of the current program such that: '), nltab,
   write('    - All heads of clauses '), write(Defs),
   write(' are unifiable to A'),
   nltab, write('    - No other head in Pj is unifiable to A'), nl, nl,
   write('This condition is not satisfied.'), nltab,
   (yesno('Do you want to force the folding anyway', n) -> true
   ; (msg_no_action, fail)).
msg_show_existing_laws :- nl,
   (yesno('Do you want to see the laws already available', n) -> show_laws
   ; true), nl.
msg_not_a_law :- write('This is not an expected entry.'), nl, msg_syntax_law.
msg_syntax_law :-
   nltab, write('A law is a term of the form: A1,...,An -> B1,...,Bm'),
   nltab, write('where A1,...,An, B1,...,Bm are atoms.'), nl.
msg_laws_empty :- nltab, write('There is no law available! '), nl.
msg_no_law_in_theory :- nltab, write('No law in this theory !'), nl.
msg_save_file_laws(File_Theory) :-
   nltab, write(File_Theory), write(' has been updated.'), nltab,
   write('The previous version has been saved in old_'), write(File_Theory), nl.
msg_not_a_goal :- write('This is not an expected entry.'), nl.
msg_enter_name_program(File2) :-
   write('New file name :  Final_Progs/'),  get_line(S_File),
   (S_File =[] -> nl, write('Program not saved.'), nl, fail
   ; msg_enter_name_program_ok(S_File,File2)).
msg_enter_name_program_ok(S_File,File3):-
   my_last(S_File,Head_S_File,46), append("Final_Progs/",Head_S_File,File2),
   name(Name_File2,File2),
   (file_exists(Name_File2) -> (msg_choose_another_filename(Name_File2),
   msg_enter_name_program(File3)) ; File2=File3).
msg_choose_another_filename(File) :-
   nl, write('File '), write(File), write(' already exists. '),
   nl,nl, write('Choose another name.').
msg_save_current_program(File) :- nltab, write('Program saved in '), write(File).
msg_save_current_theory(File) :- nltab, write('Theory saved in '), write(File).
```

```
msg_ask_save_history(File1,File2) :-
   nltab, name(Name_File1,File1), write('The history will be saved in '),
   write(Name_File1), nltab,
   (yesno('Do you agree (enter "no" to give another name) ', y) -> File1=File2
   ; msg_enter_name_history(File1,File2)).
msg_enter_name_history(File1,File2) :-
    nltab, write('New file name :  '), get_line(S_File),
    (S_File = [] -> File1=File2, msg_default_name_file
    ; (my_last(S_File,Head_S_File,46), append("Histories/",Head_S_File,File2))).
msg_default_name_file :-
    nltab, write('No selection -> default name is taken.'), nl.
msg_save_history(ProgN, File) :-
    nltab, name(FileProgN,ProgN),write('History of program '),
    write(FileProgN), write(' saved in '), write(File), nl.
msg_ask_save_session(File1,File2) :-
   nltab, name(Name_File1,File1), write('The above session will be saved in '),
   write(Name_File1), nltab,
   (yesno('Do you agree (enter "no" to give another name) ', y) -> File1=File2
   ; msg_enter_name_session(File1,File2)).
msg_enter_name_session(File1,File2) :-
    nltab, write('New file name :  Sessions/'), get_line(S_File),
    (S_File = [] -> File1=File2, msg_default_name_file
    ; (my_last(S_File,Head_S_File,46), append("Sessions/",Head_S_File,File2))).
msg_save_current_session(File) :-
    nltab, write('Current session saved in '), write(File).
msg_not_a_clause :- write('This is not an expected entry.'), nl,
    write('Enter a term of the form "A :- B1,...,Bn.".'), nl.
msg_give_modes(P,N) :-
  nl, write('Give mode for '), write(P/N),
  write(' (+ or - for each argument):'), nl.
msg_no_available_progs :-
  nltab, write('No available programs (stored in subdirectory Progs/) '), nl.
msg_no_available_final_progs :-
  nltab, write('No programs in subdirectory Final_Progs/) '), nl.
msg_no_available_theories :-
  nltab, write('No available theories (stored in subdirectory Theories/) '), nl.
msg_no_available_sessions :-
  nltab, write('No available sessions (stored in subdirectory Sessions/) '), nl.
msg_undefined_predicate(Pred) :-
  nltab, write('Warning: undefined predicate '), write(Pred), nl.
msg_not_in_ancestor([Cl]) :- !, nltab,
  write('Clause '), write(Cl),
  write(' is not in an ancestor of current program.'), nl.
msg_not_in_ancestor(Cls) :- nltab, write('Clauses '), write(Cls),
  write(' are not in an ancestor of current program.'), nl.
msg_helps :-
        show_list_v([
        'st.            < start a new transformation process >',
        'r.             < show available rules >',
        's.             < show available strategies >',
        'c.             < show available other commands >',
        'helps.         < show this five lines menu >' ]), nl, nl.
```

```
ask_enter_definition(H,B) :-
     nl, write('enter definition clause: '), nl, read(D2),
     check_clause(D2), rearrange_definition(D2, (H,B)).
ask_enter_first_definition_clause(D,Head,Body) :-  nl,
  (yesno('Do you want to check the existing definitions', n) -> show_definitions
  ; true), nl,
  write('enter clauses (use the same new predicate in the heads): '), nl,
  write('first clause: '), read(D2), nl, check_clause(D2),
  rearrange_definition(D2,(Head,Body)), D = (Head,Body),
  check_head_variable(Head,Body).
ask_enter_next_definition_clause(L,Functor_Head,Arity,Defs) :-
  (yesno('Another definition',n) -> (write('clause: '), read(D2),
  check_clause(D2),
  rearrange_definition(D2,(Head,Body)), D = (Head,Body),
  (predname_and_arity(Head,Functor_Head,Arity) -> append(L,[D], L2)
  ; (write(' The head of the clause must use predicate '),
  write(Functor_Head/Arity), L2=L) ; true), nl,
  ask_enter_next_definition_clause(L2,Functor_Head,Arity,Defs)) ; L=Defs).
ask_initial_position(Length,Pos) :-
  ask_number('initial position ',1,Length, 1,Pos).
ask_final_position(Length,Pos) :- ask_number('final position ',1,Length, 1,Pos).
ask_position_of_new_goal(Length,Pos) :-
  Length1 is Length + 1,
  ask_number('position of new goal',1,Length1, Length1,Pos).
ask_law_number([],_) :- msg_laws_empty, fail.
ask_law_number(Defs,Ans) :-
  ask_oneof('law number ',Defs,no,Ans),
  (Ans=[no] -> msg_failure_no_selection ; true).
ask_definition_number([],_) :- msg_definition_clauses_empty, fail.
ask_definition_number(Defs,Ans) :-
  ask_oneof('definition number ',Defs,no,Ans), (Ans=no -> msg_no_action ; true).
ask_enter_goal_to_add(L) :-
  nl, write('Goal: '), read(Goal), nl, check_goal(Goal), conj_to_list((Goal),L).
ask_enter_replacing_clause(C) :-
  nl, write('replacing clause: '), nl, read(C), check_clause(C), nl.
ask_enter_clause_to_add(C) :-
   nl, write('enter new clause: '), nl, read(C), check_clause(C), nl.
ask_enter_law(Law) :-
   nl, write('enter new law: '), nl, read(Law), nl.
ask_position_introduction(CNs,Pos) :-
   ask_oneof('at which position ?',CNs,'at last position',Pos).
ask_clause_number(CNs,Ans) :-
   ask_oneof('clause number ',CNs,no,Ans),
   (Ans=[no] -> msg_failure_no_selection ; true).
ask_position_to_unfold(Length,Pos) :- ask_number('Number ',1,Length, 1,Pos).
ask_program_for_unfolding(PN,Ancestors,Pi) :-
   ask_oneof('Program number',Ancestors,PN,Pi).
ask_positions_in_body(Length,Atom_Numbers) :-
  ask_chars('atom numbers [no]',0,100,Answer),
  (Answer=[] -> msg_no_selection, fail
  ; recover_positions(Answer,Atom_Numbers),
  check_input_atoms(Length,Atom_Numbers)).
```

```
recover_positions(Answer,Pos) :- recover_positions([],Answer,Pos).
recover_positions([],[32|Answer],Pos) :-!, recover_positions([],Answer,Pos).
recover_positions(Pref,[32|Answer],[Pos|Pos1]) :-
   name(Pos,Pref), recover_positions(Answer,Pos1).
recover_positions(Pref,[44|Answer],[Pos|Pos1]) :-
   name(Pos,Pref), recover_positions(Answer,Pos1).
recover_positions(Pref,[C|Answer],Pos) :-
   49 =< C, C =< 57, !, append(Pref,[C],Pref2),
   recover_positions(Pref2,Answer,Pos).
recover_positions(Pref,[_|Answer],Pos) :- recover_positions(Pref,Answer,Pos).
recover_positions([],[],[]) :- !.
recover_positions(Pref,[],[Pos]) :- name(Pos,Pref).
help_trace_session :-
     nl, tab(5), write('Available choices:'), nl,
     nl, tab(5), write('p -> show current program'), nl,
     nl, tab(5), write('d -> show current defs'), nl,
     nl, tab(5), write('c -> insert comment in history'), nl,
     nl, tab(5), write('o -> trace session off'), nl.
msg_user_successful_folding(Pi,A):-
   functor(A,NameDef,_), write(' Successful folding using definition '),
   write(NameDef), (write(' in P'), write(Pi)), nl.
msg_derivation_stopped :-
   nltab, write('Derivation stopped ! '), nl.
msg_failure_on_unfolding(E) :-
   nl, write('Unfolding of clause '), write(E),
   write(' fails because the leftmost atom is a failing goal.'), nl.
msg_determ_strat_fails :-
   nl, write('Failure of the strategy. '), nl,
   write('The program derived so far is the following:'), nl, pp.
msg_derivation_fails(E) :-
   msg_no_clause_deletion(E), write('The whole derivation fails.'), fail.
msg_no_clause_deletion(E) :-
   nltab, write('No deletion of clause '), write(E), write('. ').
% Case-split (global analysis)
new_clauses(case_split,Cl,Cls) :-
   integer(Cl), !, new_clauses(Cls,NewCls), nl,
   write('by case splitting clause '),
   write(Cl), write(' we get:'), nl, show_numb_clauses(NewCls), nl.
% Case-split (global analysis)
new_clauses(case_split,Cls,Ps) :-
   write('by splitting clauses '), write(Cls), write(' we get:'), nl,
   show_packets(Ps), nl.

% Case-split (binary)
new_clauses(case_split,C,C2,CN) :-
   new_clauses(CN,NewCls), nl, write('by case splitting clause '),
   write(C), write(' w.r.t. clause '), write(C2), write(' we get:'),
   nl, show_numb_clauses(NewCls), nl.
new_clauses(unfold,C,A,CN) :-
   new_clauses(CN,NewCls), nl, write('by unfolding clause '), write(C),
   write(' w.r.t. atom '), write(A), write(' we get:'), nl,
   show_numb_clauses(NewCls), nl.
```

```
new_clauses(unfold_d,C,A,CN) :-
   new_clauses(CN,NewCls), nl, write('by unfolding clause '), write(C),
   write(' w.r.t. atom '), write(A), write(' we get:'), nl,
   show_numb_clauses(NewCls), nl.
new_clauses(unfold_d_p,C,A,CN) :-
   new_clauses(CN,NewCls), nl, write('by unfolding clause '), write(C),
   write(' w.r.t. atom '), write(A), write(' using current program '),
   write(' we get:'), nl, show_numb_clauses(NewCls), nl.
new_clauses(unfold_d_i,Pi,C,A,CN) :-
   new_clauses(CN,NewCls), nl, write('by unfolding clause '), write(C),
   write(' w.r.t. atom '), write(A), write(' using program '), write(Pi),
   write(' we get:'), nl, show_numb_clauses(NewCls), nl.
% new_clauses(fold,C,As,Def_Number,CN): it
% shows the new clause CN obtained by a folding step.
new_clauses(fold,C,As,Def,CN) :-
   CN=[N], clause3(N,Cl,_), nl, write('by folding clause '), write(C),
   write(' w.r.t. atoms '), write(As), write(' using '), write(Def),
   write(' we get:'), nl, show_numb_clauses([(N,Cl)]), nl.
new_clauses(fold,Cs,Def_Numbers,CN) :-
  CN=[N], clause3(N,Cl,_), nl, write('by folding clause '),
  write(Cs), write(' using '),
  write(Def_Numbers), write(' we get:'), nl, show_numb_clauses([(N,Cl)]), nl.
new_clauses([],[]).
new_clauses([N|Ns],[(N,C)|L]) :-  clause3(N,C,_), new_clauses(Ns,L).
new_clause(delclause,C) :- nl, write('deletion of clause '), write(C), nl.
new_clause(elimclauses,Cls) :-
   nl, write('elimination of transient clause(s) '), write(Cls), nl.
new_clause(delclause_failure_body,C) :-
   nl, write('deletion of clause '), write(C),
   write(' because of failure in body'), nl.
new_clause(unfold_facts,C,As,NewCN) :-
   NewCN=[N], clause3(N,Cl,_), nl, write('by unfolding clause '), write(C),
   write(' w.r.t. atom(s) '), write(As), write(' we get:'), nl,
   show_numb_clauses([(N,Cl)]), nl.
new_clause(simpl_eq,C,As,NewCN) :-
   NewCN=[N], clause3(N,Cl,_), nl,
   write('by simplifying (in)equality(ies): '), write(As),
   write(' in clause '), write(C), write(' we get:'), nl,
   show_numb_clauses([(N,Cl)]), nl.
new_clause(ud_eq,C,As,NewCN) :-
   NewCN=[N], clause3(N,Cl,_), nl, write('by unfolding clause '), write(C),
   write(' w.r.t. (in)equalities): '), write(As), write(' we get:'),
    nl, show_numb_clauses([(N,Cl)]),nl.
new_clause(addgoal,C,Pos,L,NewCN) :-
   nl, NewCN=[N], clause3(N,Cl,_), write('by adding goal '),
   numbervars(L,0,_), show_goal(L),
   write(' in clause '), write(C), write(' at position '), write(Pos),
   write(' we get:'), nl, show_numb_clauses([(N,Cl)]), nl.
new_clause(addclause,NewCN) :-
   nl, NewCN=[N], write('by adding clause '), write(N),
   write(' we get the new program: '),  nl, show_current_program.
```

```
new_clause(moveclause,N) :-
   nl, write('by moving clause '), write(N),
   write(' we get the new program: '),  nl,  show_current_program.
new_clause(delgoal,C,As,NewCN) :-
   nl, NewCN=[N], clause3(N,Cl,_), write('by deleting goal '), write(As),
   write(' in clause '), write(C), write(' we get:'), nl,
   show_numb_clauses([(N,Cl)]), nl.
new_clause(replace_goal,C,OldGoal,Law,CN) :-
   nl, CN=[N], clause3(N,(H,B),_), law(Law,(L,R)), OldGoal=L,
   numbervars((OldGoal,R),0,_), nl,
   write('by replacing goal '), write(OldGoal), nl, write('by '), write(R), nl,
   write('in clause '), write(C), write(' using '), write('law '), write(Law),
   write(' we get:'), nl, show_numb_clauses([(N,(H,B))]), nl.
new_clause(moveatom,C,Pos1,Pos2,CN) :-
   CN=[N], clause3(N,Clause,_), nl, write('by moving atom '), write(Pos1),
   write(' to position '), write(Pos2), write(' in clause '), write(C),
   write(' we get:'), nl, show_numb_clauses([(N,Clause)]), nl.
/*-------
write_inf/[1,3] shows the history.
--------*/
write_inf(1) :-  write('P'), write('1: '), file_input(F), write(F).
write_inf(P,(C,(unf,A),_),P1) :-
   write('P'), write(P1), write(': '), write('by unfolding '), write('cl '),
   write(C), write(' in P'), write(P), write(' wrt atom '), write(A),
   write(' using P1').
write_inf(P,(C,(uunf,A),_),P1) :-
   write('P'), write(P1), write(': '), write('by unfolding '), write('cl '),
   write(C), write(' in P'), write(P), write(' wrt atom '), write(A),
   write(' using P1').
write_inf(P,(C,(uunf_i,P_j,A),_),P1) :-
   write('P'), write(P1), write(': '), write('by unfolding '), write('cl '),
   write(C), write(' in P'), write(P), write(' wrt atom '), write(A),
   write(' using P'), write(P_j).
write_inf(P,(C,(unfold_facts,As),_),P1) :-
   write('P'), write(P1), write(': '), write('by unfolding '), write('cl '),
   write(C), write(' in P'), write(P), write(' wrt atoms: '), write(As),
   write(' using P1').
write_inf(P,(C,(simpl_eq,As),_),P1) :-
   write('P'), write(P1), write(': '), write('by simplifying in '), write('cl '),
   write(C), write(' in P'), write(P), write(' (in)equalities :'), write(As).
write_inf(P,(C,(ud_eq,As),_),P1) :-
   write('P'), write(P1), write(': '), write('by unfolding '), write('cl '),
   write(C), write(' in P'), write(P), write(' wrt (in)equalities: '), write(As).
write_inf(P,(C,(fold,As,D),_),P1) :-
   write('P'), write(P1), write(': '), write('by folding '), write('cl '),
   write(C), write(' in P'), write(P), write(' w.r.t atoms '), write(As),
   write(' using '), write(D).
write_inf(P,([C],(fold,[Def]),_),P1) :- !,
   write('P'), write(P1), write(': '), write('by folding '), write('cl '),
   write(C), write(' in P'), write(P), write(' using cl '), write(Def).
```

```
write_inf(P,(Clauses_Number,(fold,Def),_),P1) :-
   write('P'), write(P1), write(': '),
   write('by folding '), write('cls '),
   write(Clauses_Number), write(' in P'), write(P),
   write(' using cls '), write(Def).
write_inf(P,(C,(case_split,Cl_Number),_),P1) :-
   write('P'), write(P1), write(': '),
   write('by case splitting '), write('cl '), write(C),
   write(' in P'), write(P), write(' wrt clause '), write(Cl_Number).
write_inf(P,(Cl,case_split,_),P1) :-
   integer(Cl), write('P'), write(P1), write(': '),
   write('by case splitting '), write('cl '),
   write(Cl), write(' in P'), write(P).
write_inf(P,(Cls,case_split,_),P1) :-
   write('P'), write(P1), write(': '), write('by case splitting '),
   write('cls '), write(Cls), write(' in P'), write(P).
write_inf(P,(C,(rep,As,Law),_),P1) :-
   write('P'), write(P1), write(': '), write('by replacing '),
   write('atoms '), write(As), write(' in '), write('cl '), write(C),
   write(' in P'), write(P), write(' using law '), write(Law).
write_inf(P,(def,[NewCN]),P1) :-
   write('P'), write(P1), write(': '), write('by adding def '),
   write(NewCN), write(' to P'), write(P).
write_inf(P,(gen_def,[NewCN]),P1) :- !,
   write('P'), write(P1), write(': '), write('by adding def '),
   write(NewCN), write(' to P'), write(P).
write_inf(P,(gen_def, L), P1) :-
   write('P'), write(P1), write(': '), write('by adding defs '),
   write(L), write(' to P'), write(P).
write_inf(P,(C,(delgoal,As),_),P1) :-
   write('P'), write(P1), write(': '), write('by deleting goal '),
   write(As), write(' in cl '), write(C), write(' in P'), write(P).
write_inf(P,(C,(addgoal,_,G),_),P1) :-
   write('P'), write(P1), write(': '),
   write('by adding goal '), numbervars(G,0,_),
   show_body(G), write(' in cl '), write(C), write(' in P'), write(P).
write_inf(P,(addclause,[N]),P1) :-
   write('P'), write(P1), write(': '),write('by adding cl '),
   write(N), write(' in P'), write(P).
write_inf(P,(_,addclause,[N]),P1) :-
   write('P'), write(P1), write(': '), write('by adding cl '),
   write(N), write(' in P'), write(P).
write_inf(P,(moveclause,[N]),P1) :-
   write('P'), write(P1), write(': '), write('by moving cl '),
   write(N), write(' in P'), write(P).
write_inf(P,(NN,repclause,[N]),P1) :-
   write('P'), write(P1), write(': '), write('by replacing cl '),
   write(NN), write(' by cl '), write(N), write(' in P'), write(P).
write_inf(P,(C,delclause,[]),P1) :-
   write('P'), write(P1), write(': '), write('by deleting cl '),
   write(C), write(' in P'), write(P).
```

```
write_inf(P,(N,loopabs),P1) :-
   write('P'), write(P1), write(' : '),
   write('by loop absorption strategy from clause '),
   write(N), write(' in P'), write(P).
write_inf(P,(N,loopabs),P1) :-
   write('P'), write(P1), write(': '),
   write('by generalization strategy, to eliminate '),
   nl, write('unnecessary variables in clause '), write(N),
   write(' of P'), write(P).
write_inf(P,(Cls,elimclauses,[]),P1) :-
   write('P'), write(P1), write(': '),
   write('by deleting transient clauses '),
   write(Cls), write(' in P'), write(P).
write_inf(P,(C,(moveatom,N,M),_),P1) :-
   write('P'), write(P1),
   write(': by moving atom from position '), write(N),
   write(' to position '), write(M), write(' in cl '), write(C),
   write(' in P'), write(P).
```

C.9 struct.pl

```
% struct.pl
%
% This file contains predicates which manipulate the data structures.
%
% A transformation tree is represented by terms of the form node(N,M,S,R,L),
% where N is a node number (1 for the root), M is the node number of the father
% of N (0 for the father's root), S is the list of the node numbers which are
% sons of M, R is a term identifying the rule applied on the program at node M
% to produce the program at node N (0 for the root), and L is the list of the
% clauses' numbers in the program at node N. A clause is represented by a term
% of the form clause3(N,C,Aux), where C is a term identifying a clause, N is a
% clause number, and Aux a parameter set to 'prog' for alll clauses in the
% intitial program. C is of the form (Hd,Bd) where Hd and Bd are an atom and a
% list of atoms respectively.

:- use_module(library(lists), [ append/3, member/2 ]).
:- use_module(library(terms), [ variant/2, unify_with_occurs_check/2 ]).

label_arc(Node,LA,SN) :- node(SN,Node,_,LA,_).
label_arc(Node,LA) :- node(Node,_,_,LA,_).
label_node(Node,LN) :- node(Node,_,_,_,LN).
type_pnat(N) :- integer(N), N>0.
type_clause(H) :- \+(H=(_:-_)), \+(number(H)), (type_list(H) -> H=[];true).
type_clause(C) :-
   C=(Head:-Body), \+(number(Head)),
   (type_list(Head) -> Head=[];true), type_goal((Body)).
type_goal(G) :- \+(G=(_,_)), \+(number(G)), (type_list(G) -> G=[];true).
type_goal(G) :-
   G=(A,B), \+(number(A)), nonvar(A), (type_list(A)->A=[];true), type_goal(B).
type_list(X) :- X ==[].
type_list(X) :- nonvar(X), X=[_|T], type_list(T).
type_law(L) :- L=(LL->RL), type_goal(LL), type_goal(RL).
clause_number_to_clause(N,C) :- clause3(N,C,_).
predname_and_arity(Head,Pred_Head,Arity) :- functor(Head,Pred_Head,Arity).
head_of_clause(C,Head) :-    clause3(C, (Head,_), _).
body_of_clause(C,Body) :-    clause3(C, (_,Body), _).
is_a_unit_clause(C) :- body_of_clause(C,[]).
current_clauses(Cls) :- current_prog(_,Cls).
current_program(P) :- current_prog(P,_).
number_clause(Num) :- number(Num).
length_body_clause(Cl,L) :- body_of_clause(Cl,B), length(B,L).
check_exists_prog_in_transf_tree(P) :- label_node(P,_), !.
check_exists_prog_in_transf_tree(P) :- msg_unknown_program(P).
check_exists_transf_tree :- clause(clause3(_,_,_),_), !.
check_exists_transf_tree :- msg_failure_no_selection.
check_exists_def :-
   label_arc(Prog,(def,_)), current_program(Current),
   check_ancestor(Prog,Current), !.
```

```
check_exists_def :-
   label_arc(Prog,(gen_def,_)), current_program(Current),
   check_ancestor(Prog,Current), !.
check_exists_law :- law(_,_), !.
check_exists_predicate(P/A) :-
     (var(P);var(A)), !, msg_unknown_predicate(P/A), msg_no_action, fail.
check_exists_predicate(P/A) :-
     functor(Pred,P,A), (undefined_predicate(Pred) ->
     (msg_unknown_predicate(P/A), msg_no_action, fail) ; true).
check_ancestor(N,PN) :- ancestors(PN,Ancestors), member(N,[PN|Ancestors]), !.
check_goal(G) :- type_goal(G) -> true; msg_not_a_goal, fail.
check_clause(C) :- type_clause(C) -> true; msg_not_a_clause, fail.
check_law(L) :- type_law(L) -> true; msg_not_a_law, fail.
check_clause_with_non_empty_body(Clause_Number,Length) :-
  body_of_clause(Clause_Number,B), length(B,Length), (Length > 0 -> true ;
  write('Clause '),  write(Clause_Number), write(' has an empty body !'), fail).
check_head_variable(Head,Body) :-
  (vars_subset(Head,Body) -> true ;
  write('Warning : some variable in the head does not occur in the body.'), nl).
check_input_clause(C) :-
     check_pnat(C), (clause3(C,(_,_),_) -> true ;
     write(C), write(' is not a clause number.'), fail),
     current_clauses(CNs), (member(C,CNs) -> true ;
     msg_not_a_clause_in_current_prog(C), fail).
check_pnat(N) :- type_pnat(N) -> true; msg_not_a_pnat, fail.
check_input_clauses([]) :- nltab, write('Enter a non empty list'), nl.
check_input_clauses([C]) :- !, check_input_clause(C).
check_input_clauses([C|L]) :- !, check_input_clause(C), check_input_clauses(L).
check_input_clauses(_) :-
   nltab, write('A list of clause numbers is expected !'), nl, fail.
check_input_atom(C,A) :-
   check_pnat(A), body_of_clause(C,B), length(B,L),
   ((A =< L, A >=1) -> true ;
   (L =0 -> (write('Clause '), write(C),
   write(' has an empty body ! No atom can be selected in it for unfolding'),
   fail) ; (write(A), write(' does not denote a position in the body of '),
   write(C), nl, write('It must be in [1..'), write(L), write(']'), nl, fail))).
check_input_program(Pj) :-
   current_program(PN), (PN=Pj -> true ; (check_ancestor(Pj,PN) -> true ;
   nl, write(Pj), write(' is not an ancestor of current program.'), nl, fail)).
check_input_atoms(L,[A]) :- integer(A), A =< L, !.
check_input_atoms(L,[A|As]) :- integer(A), A =< L, !, check_input_atoms(L,As).
check_input_atoms(_,[_|_]) :-!, msg_bad_positions_in_body, fail.
check_input_atoms(_,A) :- msg_must_be_a_list_of_atoms(A), fail.
check_input_clause_to_unfold(C) :-
   check_pnat(C), (body_of_clause(C,B) -> true ;
   write(C), write(' is not a clause number.'), fail),
   current_clauses(CNs), (member(C,CNs) -> true ;
   write('Clause '), write(C), write(' is not in the current program.'), fail),
   (B=[] ->
   (write('Clause '), write(C), write(' has an empty body !'), fail) ; true).
```

```
check_input_clause_to_unfold(C, Length) :-
   check_pnat(C), (body_of_clause(C,B) -> true ;
   write(C), write(' is not a clause number.'), fail),
   current_clauses(CNs), (member(C,CNs) -> true ;
   write('Clause '),  write(C), write(' is not in the current program.'), fail),
  (B=[] -> (write('Clause '), write(C), write(' has an empty body !'), fail)
  ; length(B,Length)).
check_input_clause_to_fold(C,L) :- check_input_clause_to_unfold(C,L).
check_input_clause_for_goal_rep(C,L) :- check_input_clause_to_unfold(C,L).
check_input_clause_for_move_lit(C,L) :- check_input_clause_to_unfold(C,L), L >1.
check_input_clause_for_goal_deletion(C,L) :- check_input_clause_to_unfold(C,L).
check_input_clause_to_case_split(C) :-
    check_pnat(C), (clause3(C,(_,_),_) -> true ;
    write(C), write(' is not a clause number.'), fail),
    current_clauses(CNs), (member(C,CNs) -> true ; write('Clause '),
    write(C), write(' is not in the current program.'), fail).
check_new_pred(Head,_,_) :- (check_new_pred(Head) -> true ; fail).
check_new_pred(Head_def) :- predname_and_arity(Head_def,F,Arity),
   (setof((Head,Body),N^clause3(N,(Head,Body),prog),Clauses) -> true
   ; msg_failure_no_selection), !,
   (setof(H, N_Def^Body_Def^definition(N_Def, (H,Body_Def)),Heads) -> true
   ; Heads=[]), not_in_prog(F,Arity,Clauses), not_in_def(F,Arity,Heads).
definition(N,(Head,Body)) :-
    clause3(N,(Head,Body),[def|L]), label_arc(Prog,(def,L)),
    current_program(Current), check_ancestor(Prog,Current).
definition(N,(Head,Body)) :-
    clause3(N,(Head,Body),[gen_def|L]), label_arc(Prog,(gen_def,L)),
    current_program(Current), check_ancestor(Prog,Current).
not_in_prog(F,A,Clauses) :- not_functor_ex(F,A,Clauses), !.
not_in_prog(F,A,_) :- msg_already_used_in_prog(F/A).
not_in_def(F,A,Heads) :- not_functor(F,A,Heads), !.
not_in_def(F,A,_) :- msg_already_used_in_def(F/A).
not_functor_ex(_,_,[]).
not_functor_ex(F,Arity,[C|Cs]) :-
   C=(Head,Body), functor(Head,F1,Arity1), (\+ F = F1 ; \+ Arity=Arity1),
   not_functor(F,Arity,Body), not_functor_ex(F,Arity,Cs).
not_functor(_,_,[]).
not_functor(F,Arity,[B|Bs]) :-
  functor(B,F1,Arity1), (\+ F = F1 ; \+ Arity=Arity1), not_functor(F,Arity,Bs).
body_without_inequalities([\+(A=B)|IneqsBd_C],[\+(A=B)|Ineqs],Bd_C):-!,
    body_without_inequalities(IneqsBd_C,Ineqs,Bd_C).
body_without_inequalities(Bd_C,[],Bd_C).
body_without_equalities([A=B|IneqsBd_C],[A=B|Ineqs],Bd_C):-!,
    body_without_equalities(IneqsBd_C,Ineqs,Bd_C).
body_without_equalities(Bd_C,[],Bd_C).
non_failed_body(L) :- current_clauses(L2), non_failed_body(L,L2).
non_failed_body([],_).
non_failed_body([E|R],L) :-
    member(X,L), head_of_clause(X,H), \+ \+ E = H, !, non_failed_body(R,L).
non_failed_body([E|R],L) :-
    undefined_predicate(E,L), predname_and_arity(E,F,A),
    msg_undefined_predicate(F/A), non_failed_body(R,L).
```

```
undefined_predicate(Clause_Number,Atom_Number,F/A) :-
    body_of_clause(Clause_Number,B), pos1(Atom,B,Atom_Number),
    functor(Atom,F,A), undefined_predicate(Atom).
undefined_predicate(Atom) :-
    current_clauses(L), undefined_predicate(Atom,L).
undefined_predicate(_,[]).
undefined_predicate(Pred,[X|_]) :-
    functor(Pred,F,A), functor(GenAtom,F,A), head_of_clause(X,H),
    \+ \+ GenAtom = H, !, fail.
undefined_predicate(Pred,[_|L]) :- undefined_predicate(Pred,L).
rearrange_definitions([],[]).
rearrange_definitions([D|L],[(Head_def,Body_list)|L2]) :-
   (D=(Head_def :- Body_def) -> conj_to_list((Body_def),Body_list) ;
    D=(Head_def,Body_list)), rearrange_definitions(L,L2).
rearrange_definition(D2, (Head,Body_list)) :-
   (D2=(Head:- Body) -> conj_to_list((Body),Body_list); Head=D2, Body_list=[]).
one_clause_definition(Def) :- one_clause_definition(_,Def,_).
one_clause_definition(Prog,Def,(Head,Body)) :-
   current_program(P), definition_clauses(Prog,[Def]),
   check_ancestor(Prog,P), clause_number_to_clause(Def,(Head,Body)).
definition_clauses(Prog,Def) :- label_arc(Prog,(def,Def)).
definition_clauses(Prog,Def) :- label_arc(Prog,(gen_def,Def)).
available_definitions_for_ts_fold(L) :-
   setof(N, one_clause_definition(N), L), !.
available_definitions_for_ts_fold([]).
heads_unifiable([],H,H).
heads_unifiable([C|L],H,H3) :-
   head_of_clause(C,Hd_C),
   (var(H) -> H2 = Hd_C ; anti_unify(Hd_C,H,H2)), heads_unifiable(L,H2,H3).
available_laws_for_rep(L) :- setof(N, C^law(N,C), L), !.
available_laws_for_rep([]).
is_available_law(Law) :- law(Law,_).
law_line(Law,Law2,Line) :-
   append(_,[45,62|_],Line), last(Line,Line2,46), !,
   name(Law,Line), name(Law2,Line2).
numb_fetch_clauses([],[]).
numb_fetch_clauses([NC|NCs],[(NC,C)|Cs]) :-
   fetch_clause(NC,C), numb_fetch_clauses(NCs,Cs).
fetch_clauses([],[]).
fetch_clauses([NC|NCs],[C|Cs]) :- fetch_clause(NC,C), fetch_clauses(NCs,Cs).
fetch_clause(NC,C) :- clause_number_to_clause(NC,C).
select_ancestor(N,PN) :- ancestors_inv(PN,Ancestors), member(N,[PN|Ancestors]).
ancestors(1,[]).
ancestors(Node,C) :- node(Node,F,_,_,_), ancestors(F,N), append(N,[F],C).
ancestors_inv(1,[1]).
ancestors_inv(Node,[Node|C]) :- node(Node,F,_,_,_), ancestors_inv(F,C).
get_predicates_body(L,C,L2) :-
   body_of_clause(C,Body), get_pred_arity(L,Body,L2).
get_pred_arity(L,[],L).
get_pred_arity(L,[B|Body],L2) :-
   predname_and_arity(B,P,A), (member(P/A,L) -> get_pred_arity(L,Body,L2)
   ; get_pred_arity([P/A|L],Body,L2) ).
```

```
no_other_head_unifiable(_,[],_).
no_other_head_unifiable(Clauses_Numbers2,[C|L],A) :-
   member(C,Clauses_Numbers2), !,
no_other_head_unifiable(Clauses_Numbers2,L,A).
no_other_head_unifiable(Clauses_Numbers2,[C|L],A) :-
   head_of_clause(C,H), (H=A -> fail
   ; no_other_head_unifiable(Clauses_Numbers2,L,A)).
check_equalities([],[]).
check_equalities([(Hd,Bd)|L1],[(Hd,Bd2)|L]) :-
    true_equalities(Bd,Bd2), !, check_equalities(L1,L).
check_equalities([_|L1],L2) :- check_equalities(L1,L2).
true_equalities([],[]).
true_equalities([\+ (A=B)|L],L2):-
   \+ unify_with_occurs_check(A,B), true_equalities(L,L2).
true_equalities([\+ (A=B)|_],_):- A==B, !, fail.
true_equalities([E|L],[E|L2]):- true_equalities(L,L2).
verify_instantiation_body(B1, Bd_D,Hd_D) :-
   anti_unify(B1,Bd_D,S1,S2,T), variant(Bd_D,T),
   free_variables(Hd_D,[],[],Vars_Hd_D),
   verify_instantiation_var_body(S1, S2, Vars_Hd_D).
verify_instantiation_var_body([],[],_).
verify_instantiation_var_body([X=Y|S1], [X=_|S2], Vars_Hd_D):-
    var(Y), !, verify_instantiation_var_body(S1, S2, Vars_Hd_D).
verify_instantiation_var_body([X=_|S1], [X=Z|S2], Vars_Hd_D):-
    belongs(Z,Vars_Hd_D), !, verify_instantiation_var_body(S1, S2, Vars_Hd_D).
check_list_of_positions(X,XX) :-
    get_positions(X,Positions), append([91|Positions], [93], XX).
get_positions([],[]).
get_positions([X,X2|L], [X|L2]) :-
    integer(X), X < 57, X > 48,
    integer(X2), X2 < 57, X2 > 48, !, get_positions([X2|L],L2).
get_positions([X|L], L3) :-
    integer(X), X < 57, X > 48, !,  get_positions(L, L2),
    (L2=[] -> L3=[X] ; append([X,44],L2,L3)).
get_positions([_|L], L2) :- get_positions(L, L2).
```

C.10 basic_op.pl

```
% basic_op.pl
%
% This file contains low level predicates used in other files.

:- use_module(library(system), [ directory_files/2 ]).
:- use_module(library(lists), [ append/3, memberchk/2, member/2, permutation/2 ]).
:- use_module(library(terms), [ variant/2, unify_with_occurs_check/2 ]).

% delete(L1,L2,L): L is the result of deleting each element which occurs both in
% L1 in L2.
% Ex: delete([a,b],[c,a,b,a,c,b],[c,a,c,b]) is true.
delete([],L,L).
delete([X|R],L,L1) :- del(X,L,L2), delete(R,L2,L1).
del(X,[X|L],L).
del(X,[Y|L],[Y|L1]) :- \+(X=Y), del(X,L,L1).
% delete_insert(L1,L2,L3,L4): L4 is the result of replacing in L2 the sublist
% L1 by L3.
delete_insert( [E|L], P, Ns, P1):-
  delete( L, P, P2), replace_elt_in_list(E, P2, Ns, P1).
replace_elt_in_list(E,[E|P2], Ns, P):- !, append(Ns,P2,P).
replace_elt_in_list(E,[F|P2], Ns, [F|P]):- replace_elt_in_list(E, P2, Ns, P).
member_rest(X,[X|L],L).
member_rest(X,[Y|L],[Y|L2]) :- member_rest(X,L,L2).
append(L1,L2,L3,R) :- append(L2,L3,Z), append(L1,Z,R).
append([], []).
append([L|Ls], List0) :- append(L, List1, List0), append(Ls, List1).
% This sort does not remove duplicates.
sort_list([],[]).
sort_list([A],[A]) :- !.
sort_list([A|L],L2) :- sort_list(L,L3), ord_add_elt_list(L3,A,L2).
ord_add_elt_list([], Element, [Element]).
ord_add_elt_list([Head|Tail], Element, Set) :-
   compare(Order, Head, Element),
   ord_add_elt_list(Order, Head, Tail, Element, Set).
ord_add_elt_list(<, Head, Tail, Element, [Head|Set]) :- !,
        ord_add_elt_list(Tail, Element, Set).
ord_add_elt_list(_, Head, Tail, Element, [Element,Head|Tail]).
% split(L,N,L1,E,L2): N is a positive integer. E is the element having position
% N in L, and L is the concatenation of L1, [E], and L2.
split(L,N,L1,E,L2) :- M is N-1, length1(L1,M), append(L1,[E|L2],L), nonvar(E),!.
split(L,N,L1,E,L2) :- M is N+1, split(L,M,L1,E,L2).
% length1(L,N): N is a not negative integer. L is a list made out of N variables.
length1([],0).
length1([_|L],N) :- N>0, N1 is N-1, length1(L,N1).
% vars_subset(T1,T2): each variable occurring in the term T1 occurs also in the
% term T2.
vars_subset(T1,T2) :- free_variables(T2,[],[],V1), free_variables(T1,V1,[],[]).
intersection_vars_empty(L1,L2) :- intersection_vars(L1,L2,L3), L3==[].
```

```
intersection_vars([],_,[]).
intersection_vars([E|L],L2,[E|L3]) :- belongs(E,L2), intersection_vars(L,L2,L3).
intersection_vars([_|L],L2,L3) :- intersection_vars(L,L2,L3).
% belongs(E,L): E is an element of list L.
belongs(X,[Y|_]) :- X==Y, !.
belongs(X,[_|T]) :- belongs(X,T).
% conj_to_list(S,L): S a conjunction of the form (A1,...,An), and L is the list
% [A1,...,An].
conj_to_list(X,[]) :- X==true, !.
conj_to_list(B,L) :-
   (nonvar(B), functor(B,',',_) -> (B = (B1,B2), L=[B1|H], conj_to_list(B2,H))
   ; L=[B]).
last(Last, [Head|Tail]) :- last_1(Tail, Head, Last).
last_1([], Last, Last).
last_1([Head|Tail], _, Last) :- last_1(Tail, Head, Last).
last([E],[],E).
last([F|L],[F|L2],E):- last(L,L2,E).
my_last(L1,L2,X) :- last(L1,L3,X),!, my_last(L3,L2,X).
my_last(L,L,_).
% positions_to_elts(L,Pos,L1): the elements of list L1 are all elements of list
% L, and they occur at positions Pos.
% Ex: positions_to_elts([a,b,c,d],[2,4],[b,d]) is true.
positions_to_elts(_,[],[]).
positions_to_elts(B,[H|T],[A|L]) :- pos1(A,B,H), positions_to_elts(B,T,L).
pos1(X,[X|_],1).
pos1(X,[_|L],M) :-  pos1(X,L,N), M is N+1.
% pos(E,L,N): N is the position of the first occurrence of 'E' in the list L.
% Ex: pos (E,[a,b,E,E],3) is true.
pos(X,[Y|_],1) :- X==Y, !.
pos(X,[_|L],M) :- pos(X,L,N), M is N+1.
% delete_2(L1,L2,L3): each element of list L1 is an element of list L2. L3 is
% obtained by deleting each element of L1 from L2.
% Ex: delete_2([a,X],[c,a,X,a,c,X],[c,a,c,X]) is true.
delete_2([],L,L).
delete_2([X|R],L,L1) :- del_2(X,L,L2), delete_2(R,L2,L1).
del_2(X,[Y|L],L) :- X==Y, !.
del_2(X,[Y|L],[Y|L1]) :- del_2(X,L,L1).

% del_1(E,N,L,L1): L1 is obtained by deleting the element at position N in
% list L. Ex: del_1(E,3,[E,b,E,c],[E,b,c]) is true.
del_1(X,1,[Y|L],L) :- X==Y.
del_1(X,N,[Y|L],[Y|L1]) :- N > 1, M is N-1, del_1(X,M,L,L1).
% int_var(T,TL,V): V is the list of variables occurring in TL and not occurring
% in T.
int_var(T,TL,V) :- free_variables(T,[],[],V1), free_variables(TL,V1,[],V).
del_und(X,1,[X|L],L).
del_und(X,N,[Y|L],[Y|L1]) :- del_und(X,N2,L,L1), N is N2 + 1.
conc_position(B2,1,Right,B1):- !, append(Right,B2,B1).
conc_position([E|B2],N,Right,[E|B1]):- N2 is N-1, conc_position(B2,N2,Right,B1).
my_directory_files(Dir,Subset_Files) :-
   directory_files(Dir,Files), remove_saved_versions(Files,Subset_Files).
```

```
remove_saved_versions([],[]).
remove_saved_versions([F|L],L2) :-
    name(F,NameF), last(126,NameF),!, remove_saved_versions(L,L2).
remove_saved_versions([F|L],[F|L2]) :-
    remove_saved_versions(L,L2).
non_empty_directory([E,'..'|_],[E]) :- !.
non_empty_directory([E|L],[E|LL]) :- non_empty_directory(L,LL).
/*----------------
The following predicates have been borrowed from the quintus libraries.
------------------*/

%From library(sets):
seteq(Set1, Set2) :- subset(Set1, Set2), subset(Set2, Set1).
subset([], _).
subset([Element|Elements], Set) :-
   memberchk(Element, Set),  subset(Elements, Set).
subtract([], _, []).
subtract([Element|Elements], Set, Dif) :-
   memberchk(Element, Set), !, subtract(Elements, Set, Dif).
subtract([Element|Elements], Set, [Element|Dif]) :-
   subtract(Elements, Set, Dif).

%From library(lists):
subseq([], [], []).
subseq([Head|Tail], Sbsq, [Head|Cmpl]) :- subseq(Tail, Sbsq, Cmpl).
subseq([Head|Tail], [Head|Sbsq], Cmpl) :- subseq(Tail, Sbsq, Cmpl).

%From library(anti_unify):
/* Anti-unification is the mathematical dual of unification:  given two terms
 T1 and T2 it returns the most specific term which generalises them, T.
 T is the most specific term which unifies with both T1 and T2.  A common use
 for this is in learning; the idea of using it that way comes from Gordon
 Plotkin. The code here is based on a routine called 'generalize/5' written by
 Fernando Pereira. The name was changed because there are other ways of
 generalising things, but there is nly one dual of unification.
*/
%   anti_unify(+Term1, +Term2, -Term)
%   binds Term to a most specific generalisation of Term1 and Term2.
%   When you call it, Term should be a variable.
anti_unify(Term1, Term2, Term) :- anti_unify(Term1, Term2, _, _, Term).

%   anti_unify(+Term1, +Term2, -Subst1, -Subst2, -Term)
%   binds Term to a most specific generalisation of Term1 and Term2,
%   and Subst1 and Subst2 to substitutions such that
%       Subst1(Term) = Term1
%       Subst2(Term) = Term2
%   Substitutions are represented as lists of Var=Term pairs, where
%   Var is a Prolog variable, and Term is the term to substitute for Var.
%   When you call it, Subst1, Subst2, and Term should be variables.
anti_unify(Term1, Term2, Phi1, Phi2, Term) :-
   anti_unify(Term1, Term2, [], [], Phi1, Phi2, Term).
```

```
anti_unify(Term1, Term2, Phi1, Phi2, Theta1, Theta2, Term) :-
   (nonvar(Term1), nonvar(Term2), functor(Term1, F, N), functor(Term2, F, N) ->
    functor(Term, F, N),
   anti_unify(N, Term1, Term2, Phi1, Phi2, Theta1, Theta2, Term)
   ; Term1 == Term2 -> Theta1 = Phi1, Theta2 = Phi2, Term = Term1
   ; already_covered(Phi1, Phi2, Term1, Term2, V) ->
   Theta1 = Phi1, Theta2 = Phi2, Term = V
   ;   /* Not same functor, not identical, not covered by an existing V */
    Theta1 = [V = Term1|Phi1], Theta2 = [V = Term2|Phi2], Term = V ).
anti_unify(0, _, _, SS, ST, SS, ST, _) :- !.
anti_unify(N, S, T, SS0, ST0, SS, ST, U) :-
   arg(N, S, Sn), arg(N, T, Tn), arg(N, U, Un),
   anti_unify(Sn, Tn, SS0, ST0, SS1, ST1, Un),
   M is N-1, anti_unify(M, S, T, SS1, ST1, SS, ST, U).
already_covered([V=S0|_], [V=T0|_], S, T, V) :- S0 == S, T0 == T, !.
already_covered([_|Phi1], [_|Phi2], S, T, V) :-
   already_covered(Phi1, Phi2, S, T, V).

%From library(occurs):
%   free_of_var(+Variable, +Term)
%   is true when the given Term contains no sub-term identical to the
%   given Variable (which may actually be any term, not just a var).
%   For variables, this is precisely the "occurs check" which is
%   needed for sound unification.
free_of_var(Variable, Term) :- Term == Variable, !, fail.
free_of_var(Variable, Term) :-
   compound(Term), !, functor(Term, _, Arity),
   free_of_var(Arity, Term, Variable).
free_of_var(_, _).
free_of_var(1, Term, Var) :- !, arg(1, Term, Arg), free_of_var(Var, Arg).
free_of_var(N, Term, Var) :-
   arg(N, Term, Arg), free_of_var(Var, Arg), M is N-1, !,
   free_of_var(M, Term, Var).
%   occurrences_of_var(+Term, +Variable, ?Tally)
%   is true when the given Variable occurs exactly Tally times in
%   Term.  It can only be used to calculate or test Tally;
%   to enumerate Variables you'll have to use sub_term/2 and then
%   test them with this routine.  If you just want to find out
%   whether Variable occurs in Term or not, use contains_var/2
%   or free_of_var/2.  This used to be called occ/3.
occurrences_of_var(Variable, Term, Occurrences) :-
   occurrences_of_var(Term, Variable, 0, Tally), Occurrences = Tally.
occurrences_of_var(Term, Var, SoFar, Tally) :- Term == Var, !, Tally is SoFar+1.
occurrences_of_var(Term, Variable, SoFar, Tally) :-
   compound(Term), !, functor(Term, _, Arity),
   occurrences_of_var(Arity, Term, Variable, SoFar, Tally).
occurrences_of_var(_, _, Tally, Tally).
occurrences_of_var(1, Term, Variable, SoFar, Tally) :- !,
   arg(1, Term, Argument), occurrences_of_var(Argument, Variable, SoFar, Tally).
occurrences_of_var(N, Term, Variable, SoFar, Tally) :-
   arg(N, Term, Argument), occurrences_of_var(Argument, Variable, SoFar, Accum),
   M is N-1, occurrences_of_var(M, Term, Variable, Accum, Tally).
```

```
% free_variables(+Goal, +Bound, +Vars0, -Vars) binds Vars to the union of
% Vars0 with the set of *FREE* variables in Goal, that is the set of variables
% which are captured neither by Bound nor by any internal quantifiers or
% templates in Goal.
free_variables(Term, Bound, Vars0, Vars) :-
   ( nonvar(Term) -> free_variables_1(Term, Bound, Vars0, Vars)
   ; term_is_free_of(Bound, Term), list_is_free_of(Vars0, Term) ->
   Vars = [Term|Vars0]
   ;   Vars = Vars0 ).
free_variables_1((Conjunct,Conjuncts), Bound) --> !,
   free_variables(Conjunct,  Bound), free_variables(Conjuncts, Bound).
free_variables_1((Disjunct ; Disjuncts), Bound) --> !,
   free_variables(Disjunct,  Bound), free_variables(Disjuncts, Bound).
free_variables_1(NormalGoal, Bound) --> data_variables(NormalGoal, Bound).

% data_variables(+Term, +Bound, +Vars0, -Vars) binds Vars to the union of Vars0
% with the set of variables in Term which do not occur in (are not "captured by")
% Bound. When the Bound contains no variables, it delivers the same results as
% term_variables(Term, Vars0, Vars).
data_variables(Term, Bound, Vars0, Vars) :-
   ( nonvar(Term) -> functor(Term, _, N),
   data_variables(N, Term, Bound, Vars0, Vars)
   ; term_is_free_of(Bound, Term), list_is_free_of(Vars0, Term) ->
   Vars = [Term|Vars0]
   ;   Vars = Vars0 ).
data_variables(N, Term, Bound) -->
   (  { N =:= 0 } -> [] ; { arg(N, Term, Arg), M is N-1 },
   data_variables(Arg, Bound), data_variables(M, Term, Bound)).
% term_is_free_of(+Term, +Var) is a meta-logical predicate which is true when
% the variable Var does not occur anywhere in the term Term.  It is used when
% the Term is a tree built from all the existential quantifiers and Templates
% dominating (the goal containing) this variable.
term_is_free_of(Term, Var) :-
   ( var(Term) -> Term \== Var
   ; functor(Term, _, N), term_is_free_of(N, Term, Var) ).
term_is_free_of(N, Term, Var) :-
   ( N =:= 0 -> true ; arg(N, Term, Arg), term_is_free_of(Arg, Var), M is N-1,
    term_is_free_of(M, Term, Var) ).
% list_is_free_of(+Vars0, +Var) is a meta-logical predicate which is true when
% the variable Var is not an element of Vars0, which is known to be a list of
% variables. It is used when Vars0 is the set of free variables which have been
% built up so far, and we are considering whether Var should be added to this
% set.
list_is_free_of([], _).
list_is_free_of([Head|Tail], Var) :- Head \== Var, list_is_free_of(Tail, Var).

%From library(charsio):
% chars_to_term(+Chars, ?Term) takes a list of characters such as "f(a,b)",
% lacking a terminal full stop and layout character, and reads a term from it,
% such as f(a,b).
%   It fails if there is anything left over.
```

```
chars_to_term(Chars, Term) :-
   chars_to_stream(Chars, 3, Stream), current_input(OldStream),
   set_input(Stream),
   (read(X), get0(-1) -> Flag = 1 ; Flag = 0),
   set_input(OldStream), close(Stream),
   Flag =:= 1, Term = X.
chars_to_stream(Chars, Dotp, Stream) :-
   chars(Chars, 0, Length), !, new_byte_vector(Length, Dotp, Vector),
   Vector =\= 0, chars_to_byte_vector(Chars, Vector, Length),
   DottedLength is Length+Dotp,
   stream_from_byte_vector(Vector, DottedLength, StreamCode, 0),
   stream_code(Stream, StreamCode).
chars_to_stream(Chars, _, _) :-
   format(user_error, '~N! chars_to_stream/2: bad Chars argument ~w~n', [Chars]),
   fail.
% chars(+Chars, +L0, -L) succeeds when Chars is a proper list of character
% codes, and L-L0 is its length.
chars(-, _, _) :- !, fail.
chars([], L, L).
chars([Char|Chars], L0, L) :-
   integer(Char), 0 =< Char, Char =< 255, L1 is L0+1, chars(Chars, L1, L).
%  Store the characters of Chars into the byte vector Vector. The current
% length Length is checked for overflow.
chars_to_byte_vector([], _, _).
chars_to_byte_vector([Char|Chars], Vector, Length) :-
   Length > 0,  store_byte(Char, Vector, Next), Left is Length-1,
   chars_to_byte_vector(Chars, Next, Left).

%From library(ask):
% ask_chars(+Prompt, +MinLength, +MaxLength, ?Answer) expects an answer which
% is a sequence of between MinLength and MaxLength characters, not beginning
% with a '?'.
ask_chars(Prompt, MinLength, MaxLength, Answer) :-
   repeat,
   prompted_line([Prompt,': '], Chars), length(Chars, Length),
   (   Length >= MinLength, Length =< MaxLength, \+ (Chars = [0'?|_])
    ;   format(user_output, '~NPlease enter between ~d and ~d characters
        followed by RETURN.
   Do not add a full stop unless it is part of the answer.~n',
   [MinLength,MaxLength]), fail ), !, Answer = Chars.
% ask_default_character(+Spec, ?Char) lets the programmer specify the default
% character in whatever way s/he finds convenient, either as an integer, as a
% string, or as a Prolog atom.  The case of the character is preserved.  We
% export it so your program can use the same convention elsewhere.
ask_default_character(Spec, Char) :-
   (   integer(Spec) -> Char = Spec ; atomic(Spec) -> name(Spec, [Char|_])
   ;   Spec = [Head|_], nonvar(Head), ask_default_character(Head, Char) ).
% yesno(+Question, +Default). This predicate has been changed by S. Renault wrt
% wrt its original definition in order to avoid the call to the goal to_lower.
```

```
yesno(Question, Default) :-
    ask_default_character(Default, DefChar),
    repeat,
    prompted_char([Question,' [',{DefChar},']? '], Answer),
%    to_lower(Char, Answer),
    (   is_endline(Answer), !,    (DefChar = 89 ; DefChar= 121)
%    to_lower(DefChar, 0'y)
    ;   Answer =:= "y" -> !, true
    ;   Answer =:= "Y" -> !, true
    ;   Answer =:= "n" -> !, fail
    ;   Answer =:= "N" -> !, fail
    ;   prompt('Please answer Yes or No followed by RETURN'), ttynl, fail ).
% asks a question where the answer should be an integer in the range
% Lower..Upper. If the user enters an empty line, the Default (which need not
% be in range, or even an integer) is returned instead. A typical use might be
%   ask_between('Item number', 1, N, 0, Index).
ask_between(Prompt, Lower, Upper, Default, Answer) :-
   repeat,
   prompted_numeric_line([Prompt,' [',Default,']: '], Chars),
   (   Chars = "", X = Default
   ; number_chars(X, Chars), integer(X), X >= Lower, X =< Upper
   ; format(user_output,'~NPlease enter an integer between ~d and ~d followed
      by RETURN.~n', [Lower,Upper]), fail ), !, Answer = X.
% ask_oneof(+Prompt, +Constants, +Default, ?Selected) expects an answer to be
% (a prefix of) one of the constants in the given list of constants. If the user
% enters an empty line, the Default (which need not be among the Constants) is
% returned.
ask_oneof(Prompt, Constants, Default, Selected) :-
   repeat,
   prompted_line([Prompt,' [',Default,']: '], Chars),
   (   Chars = "", X = Default
   ;   prefix_of_unique(Constants, Chars, X)
   ;   format(user_output, '~NPlease enter one of these constants~n ~p
       followed by a RETURN.
   If you just type RETURN,~n ~p will be taken as your answer.
   Do not add a full stop.~n',
   [Constants,Default]), fail ), !, Selected = X.
% prefix_of_unique(Constants, Chars, Selected) is true when (1) Chars is the
% name/2 of Selected which is a member of the list of constants Constants, or
% (2) Chars is a prefix of that name, and there is only 1 such element.
prefix_of_unique(_, [0'?|_], _) :- !, fail.
prefix_of_unique(Constants, Chars, Sel) :-
  prefix_of_unique(Constants, Chars, _, Sel).
prefix_of_unique([], _, Flag, Selected) :- !, nonvar(Flag), Selected = Flag.
prefix_of_unique([Constant|_], Chars, _, Selected) :-
   name(Constant, Chars), !, Selected = Constant.
prefix_of_unique([Constant|Constants], Chars, Flag, Selected) :-
   \+ (name(Constant,L), \+ append(Chars,_,L)), !, Flag = Constant,
   prefix_of_unique(Constants, Chars, Flag, Selected).
prefix_of_unique([_|Constants], Chars, Flag, Selected) :-
   prefix_of_unique(Constants, Chars, Flag, Selected).
```

```
% ask_number(+Prompt, +Default, ?Answer) asks a question where the answer should
% be some sort of number, either integer or float.  The answer is left as
% whatever type the user supplied:  integers are not converted to floats.
% If the user supplies an empty line, the Default (which need not be a number)
% is returned. Layout, "+", and "_" are ignored.
ask_number(Prompt, Default, Answer) :-
   repeat,
   prompted_numeric_line([Prompt,' [',Default,']: '], Chars),
   (   Chars = "", X = Default
   ;   number_chars(X, Chars)
   ;   format(user_output, '~NPlease enter a number followed by RETURN~n', []),
       fail  ), !, Answer = X.
% ask_number(+Prompt, +Lower, +Upper, +Default, ?Answer) asks a question where
% the answer should be a number in the range Lower..Upper.  If the user enters
% an empty line, the Default (which need not be in range, or even a number)
% is returned.
% A typical use might be ask_number('Percentage', 0, 100, 50, Percentage).
ask_number(Prompt, Lower, Upper, Default, Answer) :-
   repeat,
   prompted_numeric_line([Prompt,' [',Default,']: '], Chars),
   (   Chars = "", X = Default
   ;   number_chars(X, Chars), X >= Lower, X =< Upper
   ; format(user_output,'~NPlease enter a number between ~w and ~w
        followed by RETURN~n', [Lower,Upper]), fail ), Answer = X.

%From library(prompt):
% prompt(+ListOrPrompt) writes a Prompt (or a List of terms making up a Prompt)
% to the terminal, starting at the beginning of a new line, and forcing the
% output out without adding a newline.  It is used for writing prompts. Note
% that a list of character codes will NOT be printed as if it were a string.
% This may be reconsidered.
prompt(ListOrPrompt) :-
   current_output(OldTell), set_output(user_output), format('~N', []),
   prompt_(ListOrPrompt), ttyflush, set_output(OldTell).
prompt_([]) :- !.
prompt_([Head|Tail]) :- !, prompt_(Head), prompt_(Tail).
prompt_({Ch}) :- integer(Ch), Ch >= 0, Ch =< 255, !, put(Ch).
prompt_(Other) :- write(Other). % must be write, not writeq
prompted_char(Prompt, Char) :-
   prompt(Prompt),               % write the prompt to the terminal
   ttyget0(C1),                  % read the first character
   (   is_endline(C1)            % either it was an end of line
   ;   repeat,                  % or else we have to skip the
   ttyget0(C2),                  % rest of the terminal line
   is_endline(C2)), !, Char = C1. % we have to consume the end of line
% prompted_numeric_line(+Prompt, ?Chars) is a version of prompted_line/2 which
% strips away layout characters, underscores, and leading "+" signs from its
% result.  It also strips any "."s which are not followed by a digit, so that
% it does no harm to enter "123." as if you were typing to read/1.
prompted_numeric_line(Prompt, Neat) :-
   prompted_line(Prompt, Chars), filter_numeric(Chars, 0, Neat).
```

```
filter_numeric([], _, []).
filter_numeric([Char|Raw], State, Neat) :-
   (   Char =< " " -> true
   ;   Char =:= "_" -> true
   ;   Char =:= "+" -> State =:= 0
   ;   Char =:= "." -> not_before_a_digit(Raw)
   ;   fail ), !, filter_numeric(Raw, State, Neat).
filter_numeric([Char|Raw], _, [Char|Neat]) :- filter_numeric(Raw, 1, Neat).
not_before_a_digit([]).
not_before_a_digit([X|_]) :- ( X < "0" -> true ; X > "9" -> true ).
% prompted_line(+Prompt, -Chars) writes a prompt to the terminal and reads a
% line of characters from the terminal.  It fails if the user ended the line
% with the EOF character. This might move into ask.pl eventually.
prompted_line(Prompt, Chars) :- prompt(Prompt), fget_line(user, Chars).
% fget_line(+Stream, ?Chars) reads a line from the given input Stream, and
% returns the characters in the list Chars.  It does NOT return the line
% terminating character, so it is useful for portable programming.  If the
% terminator was the end of the file, it simply fails and later calls will abort.

%From library(lineio):
fget_line(Str, Chars) :-
   fget_line(Str, Line, Terminator), Terminator >= 0, Chars = Line.
% fget_line(+Stream, ?Chars, ?Terminator) reads a line from the given input
% Stream, and returns the characters in the list Chars, and the line terminating
% character in Terminator. If the terminator was end of file, it just returns
% it like always. When you use this routine, the last line will often be ignored
%if not properly ended.
fget_line(Stream, Chars, Term) :-
   current_input(OldInput), see(Stream), get_line_1(Line, Terminator),
   set_input(OldInput), Chars = Line, Term = Terminator.
get_line_1(Line, Terminator) :-      % 9 is the ASCII TAB character.
   get0(Char),                       % 5 is the EBCDIC TAB character.
   (   Char < " ", Char =\= 9, Char =\= 5 ->
       Line = [], Terminator = Char
   ;   Line = [Char|Chars], get_line_1(Chars, Terminator) ).
% get_line(?Chars) reads a line from the current input stream, and returns the
% characters in the list Chars. It does NOT return the line terminating character,
% so it is useful for portable programming. If the terminator was the end of the
% file, it simply fails and later calls will abort.
get_line(Chars) :- get_line_1(Line, Terminator), Terminator >= 0, Chars = Line.

%From library(getfile):
get_file(File, Lines) :-
   current_input(OldStream), open(File, read, NewStream), set_input(NewStream),
   get_file(Ls), set_input(OldStream), close(NewStream), Lines = Ls.
get_file(Lines) :-
   get0(Char),
   (   Char  <   0 -> Lines = []
   ;   Char =:= 10 -> Lines = [[]|Rest], get_file(Rest)
   ;                  Lines = [Line|Rest], get_file(Char, Line, Rest) ).
```

```
get_file(Char, [Char|Line], Lines) :-
   get0(Next),
   (   Next  >  10 -> get_file(Next, Line, Lines)
   ;   Next =:= 10 -> Line = [], get_file(Lines)
   ;   Next  <   0 -> Line = [], Lines = []
   ;                  get_file(Next, Line, Lines) ).

%From library(printlength):
print_length(Command, Length) :-
   open_null_stream(Stream), 'print length'(Command, user, Stream),
   character_count(Stream, CharsWritten), line_count(Stream, LineNumber),
   close(Stream),
% LineNumber = 1,
   LineNumber = 0,      %%%%% Quintus -> Sicstus
   Length = CharsWritten.
'print length'(Module:Command, _, Stream) :- !,
  'print length'(Command, Module, Stream).
'print length'(write(X), _, Stream) :- !, write(Stream, X).
'print length'(writeq(X), _, Stream) :- !, writeq(Stream, X).
'print length'(display(X), _, Stream) :- !, write_canonical(Stream, X).
'print length'(write_canonical(X), _, Stream) :- !, write_canonical(Stream, X).
'print length'(print(X), _, Stream) :- !, print(Stream, X).
'print length'(format(F,L), _, Stream) :- !, format(Stream, F, L).
'print length'(Command, Module, Stream) :-
   current_output(OldOutput), set_output(Stream),
   (   Module:Command -> true
   ;   set_output(OldOutput), close(Stream), fail ), set_output(OldOutput).

%From library(ebctypes):
% is_endline(+Char) succeeds when Char is the code of some character which
% terminates a line.  Char must already be instantiated.
is_endline(C) :-                % line terminator, NOT just newline
   C < 32,                      % covers EOF, ^G, ^J, ^K, ^L, ^M, ^Z, ESC, ^_
   C =\= 9 /* ^I */.            % not ^I (TAB) though.
```

C.11 environment.pl

```
% environment.pl
%
% This file contains the predicates that are responsible for the updating of
% the derivation tree when a transformation step is performed, and of other
% information necessary for performing further transformation steps.
% The tree is represented as a set of facts of the form:
% node(Node,Father,Sons,LA,LN), where: Node is positive integer identifying a
% node, Father is not negative integer identifying the father of Node, Sons is
% a list of positive integers identifying the sons of Node, LA and LN are
% respectively the label of the arc to the node Node and the label of Node.

:- use_module(library(lists), [ append/3 ]).

/*-----------
 apply_start(P): Initialize a new transformation sequence with P as initial
 program.
-----------*/
apply_start(P) :-
    erase_environment, assert(file_input(P)), see(P), read_list(1), seen,
    show_initial_program, init_environment.
/*-------------
 erase_environment: Empty current environment.
 init_environment: Initialize a transformation tree.
--------------*/
erase_environment :-
   retractall(file_input(_)), retractall(new_file_input(_)),
   retractall(node(_,_,_,_,_)), retractall(maxnode(_)),
   retractall(max_law(_)), retractall(current_prog(_,_)),
   retractall(law(_,_)), retractall(max_clause(_)),
   retractall(number_clauses(_)), retractall(clause3(_,_,_)).
init_environment :-
    setof(N,C^clause3(N,C,prog),LN), length(LN,L),
    assert(max_clause(L)), assert(max_law(0)), assert(current_prog(1,LN)),
    asserta(node(1,0,[],0,LN)), asserta(maxnode(1)).
/*-------------
 apply_set_current_program(NP): Set current program to NP. This predicate is
 used by set_current_program at top level.
------------*/
apply_set_current_program(NP):-
   label_node(NP,L), current_prog(N,NCs), retract(current_prog(N,NCs)),
   assert(current_prog(NP,L)).
/*-------------
 apply_add_law(Law): Add a new replacement law Law in the current theory.
------------*/
apply_add_law(Law) :-
    Law=(Left->Right), conj_to_list((Left),Left_List),
    conj_to_list((Right),Right_List),
    (clause(max_law(_),_) -> max_law(N), retract(max_law(N))  ; N=0),
    M is N+1, assert(law(M,(Left_List,Right_List))), assert(max_law(M)).
```

```
/*-------------
 apply_delete_law(Law): Delete a replacement law Law from the current theory.
------------*/
apply_delete_law(N) :-
    law(N,L), retract(law(N,L)), retract(max_law(K)),
    K1 is K-1, assert(max_law(K1)).
/*---------------
 read_list(N): Read a file which contains a prolog program, and assert each
 clause of the program in the environment using the dynamic predicate clause3.
-----------------*/
read_list(N) :- read(C), C\==end_of_file, !, (C=(Head :- Body) ->
    (conj_to_list(Body, List_Lit) -> assert(clause3(N,(Head,List_Lit),prog));
     (Body==true -> assert(clause3(N,(Head,[]),prog))
     ; assert(clause3(N,(Head,[Body]),prog)))) ;
     assert(clause3(N,(C,[]),prog))), M is N+1, read_list(M).
read_list(N) :- M is N-1, assert(number_clauses(M)).
/*---------------
 update_current_prog((N,P),(N1,P1)): retract current program N and assert
 the new one P1.
-----------------*/
update_current_prog((N,P),(N1,P1)) :-
   retract(current_prog(N,P)), assert(current_prog(N1,P1)).
/*---------------
 add_clauses(Numbers,Clauses,Aux): the clauses in the list Clauses are added to
 the current program. They are appropriately numbered.
-----------------*/
add_clauses([],[],_).
add_clauses([M|Ms],[C|Cs],Aux) :-
     max_clause(N), M is N+1, assert(clause3(M,C,Aux)),
     retract(max_clause(N)), assert(max_clause(M)), add_clauses(Ms,Cs,Aux).
/*---------------
update_programs(LA,PN): it updates the tree of programs.
-----------------*/
%below is the case where the transformation concerns a list of clauses in
%the current program, like, for instance, the fold, delete_clauses,
%subsumption, restriction rules...
update_programs(([E|LN],R,Ns),PN) :-
    current_prog(M,P), delete_insert([E|LN], P, Ns, P1),
    expand(M,([E|LN],R,Ns),P1,PN), update_current_prog((M,P),(PN,P1)).
%below is the case where the transformation concerns one clause only in the
%current program, like, for instance, the ts_fold, rearrange_and_fold, unfold
%(with its various options), add_goal, delete_goal, move_lit, case_split,
%delete_clause, replacement_clause, replacement_goal... A special case is done
%for addclause, because an additional parameter is needed  for the position
%where to add the new clause.
update_programs(('at last position',addclause,Ns),PN) :- !,
    current_prog(M,P), append(P,Ns,P1),
    expand(M,(last,addclause,Ns),P1,PN), update_current_prog((M,P),(PN,P1)).
update_programs((N,addclause,Ns),PN) :- !,
    current_prog(M,P), pos1(N,P,Pos), split(P,Pos,BeforeNodes,N,AfterNodes),
    append(BeforeNodes,Ns,[N|AfterNodes],P1),
    expand(M,(N,addclause,Ns),P1,PN), update_current_prog((M,P),(PN,P1)).
```

```
update_programs(('at last position',moveclause,N),PN) :- !,
    current_prog(M,P), append(P,[N],P1),
    expand(M,(last,moveclause,N),P1,PN), update_current_prog((M,P),(PN,P1)).
update_programs((Pos,moveclause,N),PN) :- !,
    current_prog(M,P), pos1(Pos,P,Position),
    split(P,Position,BeforeNodes2,Pos,AfterNodes2),
    (del(N,BeforeNodes2,BeforeNodes) -> AfterNodes=AfterNodes2 ;
    (del(N,AfterNodes2,AfterNodes), BeforeNodes=BeforeNodes2 )),
     append(BeforeNodes,[N],[Position|AfterNodes],P1),
     expand(M,(Pos,moveclause,N),P1,PN), update_current_prog((M,P),(PN,P1)).
update_programs((N,R,Ns),PN) :-
    current_prog(M,P), pos1(N,P,Pos), split(P,Pos,BeforeNodes,N,AfterNodes),
    append(BeforeNodes,Ns,AfterNodes,P1),
    expand(M,(N,R,Ns),P1,PN), update_current_prog((M,P),(PN,P1)).
%below is the case where the transformation does not use any input clauses.
%It is the case for ts_define, gen_define, add_clause.
update_programs((R,N),PN) :-
      (R=def; R =gen_def ; R=addclause), current_prog(M,P), append(P,N,P1),
      expand(M,(R,N),P1,PN),update_current_prog((M,P),(PN,P1)).
/*----------
 expand(Node,LA,LN,SNode): SNode is added to the tree as a successor node of
 node Node. SNode is labeled by LN and the arc (Node,SNode) is labeled by LA.
-----------*/
expand(Node,LA,LN,SN) :-
    maxnode(M), update(Node,M,SN), new_node(M,Node,LA,LN), update_maxnode.
/*----------
 update(Node,M,SN): Update the sons of the node Node after introduction of SN,
 a new successor.
-----------*/
update(Node,M,SN) :-
   node(Node,F,OldSons,LA,LN), retract(node(Node,F,OldSons,LA,LN)), SN is M+1,
   append(OldSons,[SN],NewSons), asserta(node(Node,F,NewSons,LA,LN)).
/*----------
 new_node(M,Node,LA,LN): a new successor node of Node is created, labeled by
 LN and with an ingoing arc labeled by LA.
-----------*/
new_node(M,Node,LA,LN) :- SN is M+1, asserta(node(SN,Node,[],LA,LN)).
/*----------
update_maxnode: update the numbers of the nodes in the tree.
-----------*/
update_maxnode:-
   maxnode(M), New is M+1, retract(maxnode(M)), asserta(maxnode(New)).
/*----------
concat_law(File_Theory,Law): add a new law at the end of a file of a theory.
-----------*/
concat_law(File_Theory,(L->R)) :-
   open(File_Theory,append,S), numbervars((L->R),0,_),
   write(S,L), write(S,' -> '), write(S,R), write(S, '.'), nl(S), close(S).
/*----------
assert_comment_history(P,Comment): assert comments associated with program P.
-----------*/
assert_comment_history(P,Comment) :- assert(comment_history(P,Comment)).
```

C.12 uve.pl

```
% uve.pl
%
% This program implements the procedure for the elimination of unnecessary
% variables described in:
%    Proietti M. and Pettorossi A.: Unfolding-Definition-Folding, in This Order,
%    for Avoiding Unnecessary Variables in Logic Programs.
%    Theoretical Computer Science 142(1):89-124, 1995.
% The sets InDefs and OutDefs occurring in that procedure are represented
% respectively as a set of facts of the form clause3(N,(A,[A1,...,An]),[def,1])
% and as a set of facts of the form clause3(N,(A,[A1,...,An]),[def,0]), in which
% A :- A1,...,An is the clause in C identified by the positive integer N.

:- use_module(library(lists), [ append/3 ]).
:- use_module(library(terms), [ variant/2 ]).

uve :-
       check_exists_transf_tree,
       write('input clause number: '), read(N),
       check_input_clause(N), clause3(N,(H,B),_),
       insert_definition((H,B),B), while, nl, delete_clause(N),
       write('new_program: '), nl, p.
insert_InDef(C) :- define(C,[1]).
define(D,Aux) :-
       (D=(Head_def :- Body_def) ->
             conj_to_list((Body_def),Body_list);D=(Head_def,Body_list)),
       check_head_variable(Head_def,Body_list),
       functor(Head_def,Functor_Head,Arity),
       check_new_pred(Head_def,Functor_Head,Arity), current_prog(PN,CNs),
       ((label_arc(PN,(def,NewCN),PNSucc), NewCN=[N], clause3(N,(H,B),[def|_]),
       variant((Head_def,Body_list),(H,B))) ->
             (label_node(PNSucc,CNsSucc),
              update_current_prog((PN,CNs),(PNSucc,CNsSucc));
             (add_clauses(NewCN,[(Head_def,Body_list)],[def|Aux]),
             update_programs((def,NewCN),PNSucc))).
while :- while_test -> true; (body_while, while).
while_test :- indefs(InDefs), InDefs=[].
indefs(InDefs) :- setof(N,indef(N),InDefs) -> true; InDefs=[].
indef(N) :- clause3(N,_,[def,1]).
body_while :- unfolding_step(UD), definition_steps(UD), folding_steps(UD).
unfolding_step(UD) :-
       indefs(InDefs), write('Clause Number in '), nl,
       cc(InDefs), nl, read(Clause_Number), write('Atom Number'), nl,
       read(Atom_Number), remove(Clause_Number),
       unfolding_step(Clause_Number,Atom_Number,UD1),
       not_final_clauses(UD1,UD).
not_final_clauses([NC|NCs],UD) :-
       clause3(NC,C,_), (not(failing(C)), not(body_empty(C))) ->
       (not_final_clauses(NCs,UD1), UD=[NC|UD1]); not_final_clauses(NCs,UD).
not_final_clauses([],[]).
body_empty((_,B)) :- B=[].
```

```
definition_steps([UD|UDs]) :- definition_step(UD), definition_steps(UDs).
definition_steps([]).
unfolding_step(Clause_Number,Atom_Number,UD) :-
       unfold(Clause_Number,Atom_Number),
       current_prog(PN,_), label_arc(PN,(_,_,UD)).
definition_step(E) :-
       clause3(E,(H,B),_), split_in_blocks(B,PartB),
       blocks_with_unvar((H,B),PartB,Blocks),
       (not(Blocks=[])-> insert_definitions((H,B),Blocks);true).
% blocks_with_unvar(C,Blocks,Blocks1): Blocks is a list of sets of atoms
% and Blocks1 is the list of all elements in Blocks having at least an
% unnecessary variable of the clause C.
% Ex: blocks_with_unvar((p(X,Y),[h(X),r(X,Z),q(Y)]),
% [[h(X),r(X,Z)],[q(Y)]],[[h(X),r(X,Z)]]) is true.
blocks_with_unvar((H,B),[Block|Blocks],Blocks1) :-
       blocks_with_unvar((H,B),Blocks,Blocks2),
       (un_var((H,Block)) -> Blocks1=[Block|Blocks2]; Blocks1=Blocks2).
blocks_with_unvar((_,_),[],[]).
un_var((H,B),V) :-
       free_variables(B,[],[],VB), free_variables(H,[],[],VH),
       member(V,VB), \+(belongs(V,VH)).
un_var((_,B),V) :- free_variables(B,[],[],VB), pos1(V,VB,N), pos(V,VB,M), \+(N=M).
un_var((H,B)) :- un_var((H,B),_).
% split_in_blocks(Atoms,Blocks): Blocks is the list of blocks in the
% set Atoms of atoms.
% Ex: split_in_blocks([r(X),p(X,Y),q(Y),r(Z)],[[r(X),p(X,Y),q(Y)],[r(Z)]])
% is true.
split_in_blocks([A],[[A]]).
split_in_blocks([Atom|Atoms],Blocks) :-
       split_in_blocks(Atoms,Old_Blocks), update_blocks(Atom,Old_Blocks,Blocks).
update_blocks(Atom,Blocks,NewBs) :-
       rels(Atom,Blocks,Blocks1), delete(Blocks1,Blocks,Blocks3),
       merge([[Atom]|Blocks1],Blocks2), NewBs=[Blocks2|Blocks3].
rels(Atom,[Block|Blocks],Blocks1) :-
       rels(Atom,Blocks,Blocks2),
       (rel_atom_block(Atom,Block) ->
       Blocks1=[Block|Blocks2]; Blocks1=Blocks2).rels(_,[],[]).
rel_atom_block(A,As) :- member(B,As), rel(A,B).
rel(A1,A2) :- free_variables(A1,[],[],V1), free_variables(A2,[],[],V2),
              member(X,V1), belongs(X,V2).
merge([H|[H1|T]],R) :- merge([H1|T],L), append(H,L,R).
merge([H],H).
insert_definitions((H,B),[Block|Blocks]) :-
       (test_def(H,Block) -> insert_definition((H,B),Block);true),
                             insert_definitions((H,B),Blocks).
insert_definitions((_,_),[]).
test_def(H,Block) :- defs(Ds), not_faithful_variants(H,Block,Ds).
defs(Defs) :- setof(D,N^def(N,D),Defs) -> true; Defs=[].
def(N,D) :- clause3(N,D,[def|_]).
not_faithful_variants(H,Block,[(HD,BD)|Ds]) :-
       not(faithful_variant(((H,Block),Block),((HD,BD),BD))),
       not_faithful_variants(H,Block,Ds).
```

```
not_faithful_variants(_,_,[]).
% faithful_variant((C,B),(C1,B1)): the set B of atoms in clause C is a
% faithful variant of the set B1 of atoms in clause C1.
% Ex. faithful_variant(((p(X),[h(X,Y),r(Z)]),[h(X,Y)]),
% ((q(X,Z),[h(X,Y),r(V)]),[h(X,Y)])) is true.
%faithful_variant(((H1,B1),Block1),((H2,B2),Block2)) :-
faithful_variant(((H1,_),Block1),((H2,_),Block2)) :-
       variant_set(Block1,Block2), free_variables(H1,[],[],VH1),
       free_variables(H2,[],[],VH2),
       free_variables(Block1,[],[],VB1), free_variables(Block2,[],[],VB2),
       check_vars((VH1,VB1),(VH2,VB2)).
% variant_set(T,T1): it is equal to variant_set_0.
variant_set(T,T1) :-
                  variant_split(T,VT), variant_split(T1,VT1),
                  copy_term(VT,CT), copy_term(VT1,CT1),
                  variant_sets(CT,CT1).
variant_sets([],[]).
variant_sets([H|T],L) :- find_variant_set(H,L,H1), del_2(H1,L,L1),
                      permutation(H,P), free_variables(P,[],[],V),
                      free_variables(H1,[],[],V1), variant(P,H1), V=V1,
                      numbervars((V,V1),0,_),
                      variant_sets(T,L1).
find_variant_set([A|As],[H1|T],H2) :- H1=[B|_],
                  variant(A,B) -> H1=H2; find_variant_set([A|As],T,H2).
variant_split([T|Ts],[[T|B]|Bs]) :- find_variant(T,Ts,B),
                delete_3(B,Ts,Ts1), variant_split(Ts1,Bs).
variant_split([],[]).
delete_3([],L,L).
delete_3([X|R],L,L1) :- del_3(X,L,L2), delete_3(R,L2,L1).
del_3(_,[],[]).
del_3(X,[Y|L],L1) :- X==Y, del_3(X,L,L1).
del_3(X,[Y|L],[Y|L1]) :- (X\==Y), del_3(X,L,L1).
find_variant(T,[T1|Ts],R) :- (variant(T,T1) -> R=[T1|LTs];R=LTs),
                  find_variant(T,Ts,LTs).
find_variant(_,[],[]).
check_vars((VH,[V|Vs]),(VH1,[V1|Vs1])) :-
       belongs(V,VH), belongs(V1,VH1), check_vars((VH,Vs),(VH1,Vs1)).
check_vars((VH,[V|Vs]),(VH1,[V1|Vs1])) :-
       \+(belongs(V,VH)), \+(belongs(V1,VH1)), check_vars((VH,Vs),(VH1,Vs1)).
check_vars((_,[]),(_,[])).
insert_definition((H,B),Block) :-
       link_vars((H,B),Block,Vars), create_new_atom(NewA,Vars),
       insert_InDef((NewA,Block)).
create_new_atom(NewA,Vars) :-
       create_new_def_symbol(1,NewN), NewA =.. [NewN|Vars].
create_new_def_symbol(N,NewN) :-
    name(N,L), append([110,101,119],L,NewI), name(Pred,NewI),
    (already_used_symbol(Pred) ->
           N2 is N + 1, create_new_def_symbol(N2,NewN) ; NewN = Pred).
already_used_symbol(Pred) :-
       definition(_,(Head_Def,_)), functor(Head_Def,Pred,_), !.
already_used_symbol(Pred) :- clause3(_,(Head,_),prog), functor(Head,Pred,_), !.
```

```
folding_steps([E|Es]) :- (folding_step(E)->true;true), folding_steps(Es).
folding_steps([]).
folding_step(N) :-
       clause3(N,(H,B),_), split_in_blocks(B,PartB),
       foldings(N,(H,B),(H,B),PartB).
foldings(N,(H,B),(H,B1),[Block|Blocks]) :-
       def(M,(HD,BD)), faithful_variant(((H,B),Block),((HD,BD),BD)),
       atoms(B1,Atom_Numbers,Block),
       ts_fold(N,Atom_Numbers,M), current_prog(PN,_),
       label_arc(PN,(_,_,[NewN])),
       clause3(NewN,(H2,B2),_),
       foldings(NewN,(H,B),(H2,B2),Blocks).
foldings(_,(_,_),(_,_),[]).
remove(Clause_Number) :-
       clause3(Clause_Number,C,[def,1]),
       retract(clause3(Clause_Number,C,[def,1])),
       assert(clause3(Clause_Number,C,[def,0])).
new_number(N,NN) :- test_number(N) -> (M is N+1, new_number(M,NN)); NN=N.
new_number(N) :- new_number(1,N).
test_number(N) :-
       clause3(_,(H,_),[def|_]), functor(H,F,_), name(N,L),
       append([110,101,119],L,L1), name(F,L1).
% link_vars(C,Atoms,Vs): Vs is the list of linking variables of the set
% Atoms of atoms in clause C.
link_vars((H,B),Atoms,Vs) :-
       delete_2(Atoms,B,B1),
       (setof(V,link_var(H,Atoms,B1,V),Vs) -> true; Vs=[]).
link_var(H,Atoms,B,V) :-
       free_variables(Atoms,[],[],VAs), free_variables((H,B),[],[],VCs),
       member(V,VAs), belongs(V,VCs).
% create_atom(Functor,Args,Atom): Atom is a term having 'Functor' as
% principal functor and the elements in the list Args as arguments.
create_atom(Functor,Args,Atom) :-
       length(Args,L), functor(Atom,Functor,L), insert_args(Args,1,Atom).
insert_args([A|As],N,Atom) :-
       insert_arg(A,N,Atom), M is N+1, insert_args(As,M,Atom).
insert_args([],_,_).
insert_arg(A,N,Atom) :- arg(N,Atom,A).
not(X) :- \+ X.
% failing(C): C is a failing clause.
failing((_,B)) :-
       member(A,B), \+(setof((L,R),(clause3(_,(L,R),prog),A=L),_)).
atoms(_,[],[]).
atoms(B,[H|T],[A|L]) :- pos1(A,B,H), atoms(B,T,L).
```

C.13 lap.pl

```
% lap.pl
%
% This file contains the predicates which are used by the "lap"
% Loop Absorbtion strategy. It is described in
%      Proietti, M. and Pettorossi, A.:
%      Construction of Efficient Logic Programs by Loop Absorption and
%      Generalization. Proceedings Meta'90 (M. Bruynooghe, ed.), Leuven,
%      Belgium (1990), p. 57-81.
%      and in
%      Proietti, M. and Pettorossi, A.:
%      The Loop Absorption and the Generalization Strategies for the
%      the Development of Logic Programs and Partial Deduction.
%      Journal of Logic Programming, 16(1,2):123-161, 1993.

:- use_module(library(lists), [ append/3, reverse/2 ]).
:- use_module(library(terms), [ subsumes_chk/2  ]).

lap :- check_exists_transf_tree,
       statistics(runtime,[To|_]), load(Prog1,Cl,Upred,NCl),
       mark_predicates(Upred,Pre),
       write('Enter horizontal bound: '),read(H_bound),
       write('Enter  vertical  bound: '),read(V_bound),nl,nl,
       assert(bound(H_bound,V_bound)), /* bound(Horizontal,Vertical) */
       Cl=[Head|Bd], Head=..[Nam|L], find_filename(Nam,Filename),
       tell(Filename), write_marked_proc(Prog1,Pre,Prog),
       faq([L|Bd],Prog,[],[],Defs,Pr,Pre),
       retractall(bound(_,_)), make_proc(Nam,Pr,Pr1),
       exclude_subsume(Pr1,Pr2), write_proc(Pr2), write('/*'),nl,
       scrivi_def(Defs), close(Filename),
       set_current_output(Filename,NCl),
       format('Horiz_bound was ~d,
               Vertical_bound was ~d ~n',[H_bound,V_bound]), nl,nl,
       statistics(runtime,[Tn|_]), Tmm is Tn - To,
       format('it all took ~3d sec. ~n', [Tmm]),
       message_final_prog_lap(Filename), p.
find_filename(Nam,Filename) :-
 name(Nam,L1), name('Final_Progs/lap_',L2), append(L2,L1,L), name(Filename,L).
write_clause([H]):- write(H),write('.'),nl.
write_clause([H|C]):- write(H),write(' :-'),nl,write_goals(C).
write_goals([]):-nl.
write_goals([X]):-write('    '),write(X),write('.'),nl.
write_goals([X|L]):-write('    '),write(X),write(','),nl,write_goals(L).
write_proc([]) :- nl.
write_proc([Cl1|Rcl]) :-
  copy_term(Cl1,Cl), numbervars(Cl,0,_), write_clause(Cl), write_proc(Rcl).
set_current_output(Filename,NCl) :-
   see(Filename), max_clause(MC), maxnode(N1), SN is N1+1, Pn is MC+1,
   read_list1(_,Pn,L,SN), seen, current_prog(N,P),
   expand(N,(NCl,loopabs),L,SN), update_current_prog((N,P),(SN,L)).
```

```
read_list1([C|Cs],N,L,SN) :-
   read(C), C\==end_of_file, !, L=[N|K], clau(C,N,SN),  M is N+1,
   read_list1(Cs,M,K,SN).
read_list1([_|_],N,K,_) :-
   M is N-1, max_clause(MC), retract(max_clause(MC)),
   assert(max_clause(M)), K=[], assert(number_clauses(M)).
clau(C,N,NN):- C=..[:-, Head, Body], conj_to_list((Body),Body1),
   assert(clause3(N,(Head,Body1),NN)).
clau(C,N,NN):- assert(clause3(N,(C,[]),NN)).

/* write_marked_proc
   1st arg: the list of all the procedures                          input
   2nd arg: the list of the relevant predicates                     input
   3rd arg: the list of all the procedures of relevant predicates   output
*/
write_marked_proc([],_,[]).
write_marked_proc([Proc1|Rest],Pre,[Proc1|Rest1]) :-
   Proc1 =[[Head|_]|_], functor(Head,Na,Ar),
   membert([Na,Ar],Pre), write_marked_proc(Rest,Pre,Rest1).
write_marked_proc([Proc|Rest],Pre,Rest1) :-
   write_proc(Proc), write_marked_proc(Rest,Pre,Rest1).
membert(H,[K|_]):-H==K.
membert(H,[K|T]):- H\==K, membert(H,T).
scrivi_def([]) :- write('*/').
scrivi_def([D1|Restdef]) :- numbervars(D1,0,_),write_clause(D1),
                            scrivi_def(Restdef).

/*load
  1st arg: the list of the procedures, i.e. the program             output
  2nd arg: the clause                                               output
  3rd arg: the list of the defined predicates                       output
*/
load(Prog,Cl, Pre,Numb) :-
    nl, nl, write('Current program'), nl, p,
    nl, ask_number('Enter the clause number ', 1, Numb),
    clause3(Numb,T,_), get_aioni(T,Cl), current_prog(_,L),
    load_clauses(L,L_cls,Numb), divide_in_procedures(L_cls,Prog,Pre),!.

load_clauses([Numb|P],C,Numb) :- load_clauses(P,C,Numb).
load_clauses([K|P],[C|S],Numb) :-   clause3(K,C1,_),
                                      get_aioni(C1,C), load_clauses(P,S,Numb).
%load_clauses([_],[],_).
load_clauses([],[],_).

get_aioni(Cl,C) :- Cl=..[',',Hd,Bd], (Bd ==[] -> C=[Hd] ; C=[Hd|Bd]).
divide_in_procedures([],[[]|[]],[_|[]]).
divide_in_procedures([Cl|R_cl],[[Cl|R_proc]|R_procs],[Pr1|R_pre]) :-
    Cl=[Head|_],functor(Head,N,Ar), [N,Ar]==Pr1,
    divide_in_procedures(R_cl,[R_proc|R_procs],[Pr1|R_pre]).
divide_in_procedures([Cl|R_cl],[[]|R_procs],[Pr1|R_pre]) :-
    Cl=[Head|_], functor(Head,N,Ar), nonvar(Pr1),[N,Ar] \== Pr1,
    divide_in_procedures([Cl|R_cl],R_procs,R_pre).
```

```
divide_in_procedures([Cl|R_cl],[[Cl|R_proc]|R_procs],[[N,Ar]|R_pre]) :-
    Cl=[Head|_],functor(Head,N,Ar),
    divide_in_procedures(R_cl,[R_proc|R_procs],[[N,Ar]|R_pre]).
mark_predicates(Unmarked,Marked):-
        clear_screen(5), write('LOOP ABSORPTION PROCEDURE'),nl,
        write('-------------------------'),nl,
        nl,nl, write('predicates defined :'),
        nl, write_blocks(Unmarked),nl,
        (yesno('Do you want to mark the predicates (y/n) ',n) ->
        marking(Unmarked,Marked); Marked=Unmarked).
marking(Unmarked,Marked):- nl,nl,
        write('Mark predicates with: '),nl,
        write('return or r  if you want to mark that predicate,'),nl,
        write('                 i  if you dont want to'),nl,nl,
        mark_each_predicate(Unmarked,Marked),
        prompt(ZZ,'|:  '),
        nl,nl,write('Remaining predicates'),nl,
        write_blocks(Marked),nl,nl,nl.
mark_each_predicate([],[]).
mark_each_predicate([X|Un],Ma):-write('Mark :'), write_pr(X),
        ask_chars('        (r/i)',1,1,Mark), tra(MARK,Mark),
        cont_marking(MARK,X,Un,Ma).
tra(MARK,[0'r]):- MARK='r'.
tra(MARK,[0'i]):- MARK='i'.
tra(_,[_]):- write('insert (r/i) follow by RETURN').
cont_marking(i,_,Un,Ma):-mark_each_predicate(Un,Ma).
cont_marking(r,X,Un,[X|Ma]):-mark_each_predicate(Un,Ma).
cont_marking(_,X,Un,Ma):-write('MARK     '), write_pr(X),
                        ask_chars('        (r/i)',1,1,Mark), tra(MARK,Mark),
                        cont_marking(MARK,X,Un,Ma).
read_mark(r) :- ask_chars('(r/i)',1,1,R), read1(R).
read1('r').
read1('i').
read1(_) :- write('insert (r/i) follow by RETURN'), read_mark(_).
message_final_prog_lap(Filename) :-
        nl, nl, write('Final program save in '),
        write(Filename), nl, nl.
clear_screen(0).
clear_screen(X):-nl,X1 is X-1,clear_screen(X1).
write_blocks([]).
write_blocks([B|BB]):- write('        ...'), write_pr(B), nl,write_blocks(BB).
write_pr(B):- B=[Pred,Ar], write(Pred), write('/'), write(Ar) .

/* faq
   1st arg: a pseudoclause, tclause, with as head a list of terms        input
   2nd arg: the program which consists in a list of procedures           input
   3rd arg: the list of the ancestors blocks                             input
   4th arg: the list of the definitions already found                    input
   5th arg: the list of the new definitions found                        output
   6th arg: the list of the tclauses,which are leaves of 1st             output
   7th arg: the list of the relevant predicates                          input
*/
```

```
faq([Head|Bd],Prog,Ant,Ddef,Add_def,Tclss,Pred) :-
    gather_builtins(Bd,Pred,Ir,Re),
    (Re =[] -> (Tclss = [[Head|Bd]], Add_def = [],!) ;
        (db(Re,[],Bloxx),
         distri1(Bd,Bloxx,Ir,Irr1,Sclss),
         links_care([Head|Bd],Sclss,Tsclss,Links),
         unf_T_subclauses(Tsclss,Prog,Ant,Ddef,Add_def,Tclss1,Pred),
         bagof(Tcl,make_tclss([Head|Irr1],Links,Tclss1,Tcl),Tclss),!)).
/* unf_T-subclauses
   1st arg: the list of T_subclauses  to unfold                          input
   2nd arg: the program, remember a list of procedures                   input
   3rd arg: the ancestor tloxs                                           input
   4th arg: the list of the definitions we already have                  input
   5th arg: the list of the definitions we add to the already            output
            existing ones
   6th arg: the list of lists of the tclauses, leaves of 1st             output
   7th arg: the list of the relevant predicates                          input
*/
unf_T_subclauses([],_,_,_,[],[],_).
unf_T_subclauses([Tscl1|Tsclss],Prog,Ant,Ddef,Add_def,Tclss,Pred) :-
  get_tlock(Tscl1,Pred,Tl1,[Head|Bdd],Link),
  unf_one_tlo(Tl1,Prog,Ant,Ddef,Add_def1,Tcll,Pred),
  append(Ddef,Add_def1,Ddef1),!, garbage_collect,
  unf_T_subclauses(Tsclss,Prog,Ant,Ddef1,Add_def2,Tclss2,Pred),
  ck_new_def(Tl1,Add_def1,Tcll,[Head|Bdd],Link,Tcl1),
  Tclss=[Tcl1|Tclss2], append(Add_def1,Add_def2,Add_def).
/* unf_one_tlo
   1st arg: a tlock, remember a tlock is a list of variables            input
            as first element, followed by the atoms
   2nd arg: the program                                                 input
   3rd arg: the ancestors tlocks                                        input
   4th arg: the old definitions                                         input
   5th arg: the new definitions                                         output
   6th arg: the list of the tclauses, leaveas of 1st                    output
   7th arg: the list of the relevant predicates                         input
*/
unf_one_tlo(Tl,_,_,Ddef,[],[Rat],_) :-        cfr_ls_def(Tl,Ddef,Rat),!.
unf_one_tlo(Tl,_,Ant,_,[Rdef],[Rat],_) :- cfr_ls_anc(Tl,Ant,Rdef,Rat),!.
unf_one_tlo(Tl,Prog,Ant,Ddef,Add_def,Tclss,Pred) :-
    controllo(Tl), gen_step(Tl,Gbl), copy_term(Tl,Tl1),!,
    faq(Gbl,Prog,[Tl1|Ant],Ddef,Add_def,Tclss,Pred),!.
unf_one_tlo(Tl,Prog,Ant,Ddef,Add_def,Tclss,Pred) :-
    selez(Tl,Before,Atom,After), find_proc(Atom,Prog,Proc),!,
    go_dn_brs(Proc,Before,Atom,After,Ant,Ddef,Add_def,Tclss,Pred,Prog),
    !, Tclss\==[].
/* go_dn_brs     (go down all the branches)
   1st arg: the procedure defining the predicate of the atom to         input
             unfold
   2nd arg: the part of the clause before the atom                      input
   3rd arg: the selected  atom                                          input
   4th arg: the part of the clause after the atom                       input
   5th arg: the list of the ancestors                                   input
```

```
   6th arg: the list of the definitions                                input
   7th arg: the list of the new definitions                            output
   8th arg: the list of the tclauses leaves of the tlox                output
   9th arg: the list of the relevant predicates                        input
  10th arg: the program                                                input
*/
go_dn_brs([],_,_,_,_,_,[],[],_,_).
go_dn_brs([C1|Prc],Bef,At,Aft,Ant,Ddef,Add_def,Tclss,Pred,Prog) :-
  go_dn_one(C1,Bef,At,Aft,Ant,Ddef,Add_def1,Tclss1,Pred,Prog),
  !, garbage_collect, append(Ddef,Add_def1,Ddef1),
  go_dn_brs(Prc,Bef,At,Aft,Ant,Ddef1,Add_def2,Tclss2,Pred,Prog),
  append(Add_def1,Add_def2,Add_def), append(Tclss1,Tclss2,Tclss).
/* go_dn_one  (go down one branch)
   1st arg: the program clause with which unfold the block             input
   2nd arg: the part of the block before the selected atom             input
   3rd arg: the selected atom                                          input
   4th arg: the part after the atom                                    input
   5th arg: the list of the ancestors                                  input
   6th arg: the list of the definitions                                input
   7th arg: the list of the new definitions                            output
   8th arg: the list of the tclauses, leaves                           output
   8th arg: the list of the relevant predicates                        input
   9th arg: the program                                                input
*/
go_dn_one(C,Bef,At,Aft,Ant,Ddef,Add_def,Tclss,Pred,Prog) :-
   copy_term(C,Cl), copy_term([Bef,At,Aft],[Bef1,At1,Aft1]),
   append(Bef,[At|Aft],Blo), copy_term(Blo,Bl),
   unify_head(Bef1,At1,Aft1,Cl,Unf),
   faq(Unf,Prog,[Bl|Ant],Ddef,Add_def,Tclss,Pred).
go_dn_one(_,_,_,_,_,_,[],[],_,_).
/* selez
   1st arg: a block                                                    input
   2nd arg: the part before the selected atom                          output
   3rd arg: the selected atom                                          output
   4th arg: the part after the selected atom                           output
*/
selez([Head|Bl],[Head|Before],Atom,After) :-
   special_list_depth(Bl,Max), split1(Bl,Max,Before,Atom,After).
/* special_list_depth
   1st arg: a list of atoms                                            input
   2nd arg: the hight of the hightest atom                             output
*/
special_list_depth([],0).
special_list_depth([Atom|L],Max) :-
   depth(Atom,D1), special_list_depth(L,D2), max(D1,D2,Max).
/* split
   1st arg: a list of atoms, such as the body of a block               input
   2nd arg: the heigth of the heightest atom                           input
   3rd arg: the list of atoms before the right one                     output
   4th arg: the atom selected                                          output
   5th arg: the list of atoms after the right                          output
*/
```

```
split1([Atom|Last],Max,[],Atom,Last) :- depth(Atom,M), Max=M.
split1([X|L],Max,[X|M],Atom,After) :- split1(L,Max,M,Atom,After).
depth(T,0) :- simple(T),!.
depth(T,M) :- functor(T,_,N),maxdepth(T,N,M).
maxdepth(T,1,M) :- arg(1,T,TX),depth(TX,M1),M is M1+1.
maxdepth(T,N,M) :- arg(N,T,TX), depth(TX,DX), DX1 is DX+1, N1 is N-1,
                   maxdepth(T,N1,M1), max(DX1,M1,M).
max(B,S,B):-B>=S,!.
max(_,B,B).

 /* unify_head
   1st arg: the part before the selected atom                          input
   2nd arg: the selected atom                                          input
   3rd arg: the part after the selected atom                           input
   4th arg: the clause with which the unfold will be done              input
   5th arg: the new clause                                             output
*/
unify_head(Bef,At,Aft,[Head|Bd],Nb) :-
   Head=At, !, append(Bef,Bd,S), append(S,Aft,Nb).
exclude_subsume([],[]).
exclude_subsume([[C]|L],[[C]|M]) :- exclude_subsume(L,M).
exclude_subsume([A|L],[X|N]) :- exist_subsume(A,X,L,M),
                                exclude_subsume(M,N).
exist_subsume(A,A,[],[]).
exist_subsume(A,X,[B|L],M) :- permute_furb(B,A,Bp),
                              subsumes_chk(A,Bp), exist_subsume(A,X,L,M).
exist_subsume(A,X,[B|L],M) :- permute_furb(A,B,Ap),
                              subsumes_chk(B,Ap), exist_subsume(B,X,L,M).
exist_subsume(A,X,[B|L],[B|M]):-exist_subsume(A,X,L,M).
/* find_proc
   1st arg: an atom                                                    input
   2nd arg: the list of procedures that constitute the program         input
   3rd arg: the procedure that defines the predicate of the atom       ourput
*/
find_proc(Atom,[Proc|_], Proc) :- functor(Atom,F,Ar),
                                  Proc=[Cl|_], Cl=[At|_], functor(At,F,Ar).
find_proc(Atom,[_|Prog],Proc) :- find_proc(Atom,Prog,Proc).
/* gather_builtins
   1st arg: a list of atoms                                            input
   2nd arg: the list of the relevant predicates                        input
   3rd arg: the sublist of 1st of all the atoms with non relevant      output
            predicates
*/
gather_builtins([],_,[],[]).
gather_builtins([H|B],Pred,I1,R1):-functor(H,Fu,Ar),
  (not_member([Fu,Ar],Pred)->
  (gather_builtins(B,Pred,I2,R1), I1=[H|I2]);
  (gather_builtins(B,Pred,I1,R2), R1=[H|R2])).

not_member(_,[]).
not_member(X,[A|L]):- X\==A, not_member(X,L).
```

```
/* make_tclss
   1st arg: a tclause                                                 input
   2nd arg: the list of links of the head of 1st with its former body input
   3rd arg: the list of lists of the leaves of the tlox               input
   4th arg: a tclause which is made up from appending one leaf o      output
            each tlox
*/
make_tclss(Tcl,[],[],Tcl).
make_tclss([Head|Bd],Links,[Troc1|Trocc],Tcl) :-
   one_tcl(Troc1,[Arg|Bd1]), copy_term([Arg|Bd1],[Arg2|Bd2]),
   copy_term([Links,[Head|Bd]],[[Lk|Lkk],Bef]), Lk=Arg2,
   append(Bef,Bd2,Td), make_tclss(Td,Lkk,Trocc,Tcl).
/* one_tcl
   1st arg: a list                                                    input
   2nd arg: an element of the list                                    output
*/
one_tcl([A|_],A).
one_tcl([_|B],C) :- one_tcl(B,C).
/* cfr_ls_def
   1st arg: a tlox                                                    input
   2nd arg: a list of definitions                                     input
   3rd arg: 1st folded, if the goal succeeds                          output
*/
cfr_ls_def(Bl,[D1|_],[Lv,Head]) :-  copy_term(D1,[Head|Bdd]),
   Head=..[_|Arg], faithful_instance(Bl,[Arg|Bdd],[Lv|Bd]), Bd=Bdd,!.
cfr_ls_def(Bl,[_|Lb],B1) :- cfr_ls_def(Bl,Lb,B1).
/* ck_new_def
   1st arg: a tlox                                                    input
   2nd arg: the list of the defininitions                             input
   3rd arg: the list of the tclauses, leaves of 1st                   input
   4th arg: a list, its head is the head of the tclause, its tail     input
            instead is the tail of the tclause to build
   5th arg: the links                                                 input
   6th arg: the list of tclauses, if 1st was a new defor an istance   output
            of a new def we fold, else 3rd
*/
ck_new_def(Tl1,Def,Tclss,[Head|Bdd],Link,[[Heanw,Nat|Bdnw]]) :-
   cfr_ls_def(Tl1,Def,Rtl), Rtl=[_,At], [Rtl]\==Tclss, At =..[Name|Args],
   all_different_variables(Args),!, make_proc(Name,Tclss,Pro1),
   exclude_subsume(Pro1,Proc), write_proc(Proc), copy_term(Rtl,[Tes,Nat]),
   copy_term([Head,Bdd,Link],[Heanw,Bdnw,Lknw]), Lknw = Tes.
ck_new_def(_,_,Tcl,[Head|Bdd],Link,Tcll0) :-
   put_together(Tcl,Head,Bdd,Link,Tcll0).
/* all_different_variables
   1st arg: a list of terms                                           input
   The goal succeeds if 1st is a list of variables with no repetitions
*/
all_different_variables([]).
all_different_variables([T|L]) :-
   var(T), not_member(T,L), all_different_variables(L).
```

```
/* cfr_ls_anc
   1st arg: a tlock                                                     input
   2nd arg: a list of tloxs, 1st's ancestors                            input
   3rd arg: the new definition                                          output
   4th arg: 1st folded                                                  output
*/
cfr_ls_anc(Tl,[B1|_],[Head|Bd],[L,He]) :-
   copy_term(B1,[Lv|Bd]), faithful_instance(Tl,[Lv|Bd],[L|Bdd]),
   gensym('new',Name), Head=..[Name|Lv], copy_term([Head|Bd],[He|B3]), B3=Bdd.
cfr_ls_anc(Tl,[_|Lb],D1,Rat) :- cfr_ls_anc(Tl,Lb,D1,Rat).
/* make_proc
   1st arg: the name of the new predicate                               input
   2nd arg: the list of the tclauses defining the new predicate         input
   3rd arg: the procedure defining the new predicate                    output
*/
make_proc(Name,Tclss,Proc) :-
   change_head(Name,Tclss,Proc1),rem_equ(Proc1,Proc2),
   facts_first(Proc2,Proc),!.
/* change_head
   1st arg: the name of the new predicate                               input
   2nd arg: the list of the tclauses defining 1st                       input
   3rd the list of the clauses defining 1st                             output
*/
change_head(_,[],[]).
change_head(Name,[[Ter|Bd]|Tcll],[[Head|Bd]|Cll]) :-
   Head=..[Name|Ter], change_head(Name,Tcll,Cll).
/* reorders the clauses of a procedure */
facts_first(L,M):-divide_facts_others(L,Facts,Others),
                  append(Facts,Others,M).
divide_facts_others([],[],[]).
divide_facts_others([[C]|L],[[C]|M],N):-divide_facts_others(L,M,N).
divide_facts_others([C|L],M,[C|N]):-divide_facts_others(L,M,N).
/* rem_equ
   1st arg: a list of clauses                                           input
   2nd arg: the same list of equalities without the equalities,         output
            which have already been used
   if a clause has an inequality e.g. a= b then this clause is removed
   from the procedure
*/
rem_equ([],[]).
rem_equ([Cl1|Prc1],Prc3) :-
   do_it_to_one(Cl1,Cl) -> (rem_equ(Prc1,Prc), Prc3 = [Cl|Prc]);
   rem_equ(Prc1,Prc3).
/* do_it_to_one
   1st arg: a clause                                                    input
   2nd arg: the same clause but the equalites have been used            input
*/
do_it_to_one([],[]).
do_it_to_one([At|Rest],Rest1) :-
   At =(_=_),!, call(At), do_it_to_one(Rest,Rest1).
do_it_to_one([At|Rest],[At|Rest1]) :- do_it_to_one(Rest,Rest1).
```

```
/* db
   1st arg: the list of atoms to divide in bloxs                    input
   2nd arg: intermediate value, set to [] on first call             input
   3rd arg: the list of blox wanted                                 output
*/
db([],Blox,Blox).
db([At|R],Bl1,Blox) :- add(At,Bl1,Bl2), db(R,Bl2,Blox).
add(At,Bl1,Bl2) :-
   setof(Bl,exist_var(At,Bl1,Bl),Setbl),!, riblock(Setbl,At,Bl1,Bl2).
add(At,Bl1,Bl2) :- append(Bl1,[[At]],Bl2).
exist_var(At,[Bl|_],Bl) :- var_is_in(At,Bl).
exist_var(At,[_|Bl1],Bl) :- exist_var(At,Bl1,Bl).
var_is_in(At,[C|_]) :- shared_var(At,C).
var_is_in(At,[_|Bl]) :- var_is_in(At,Bl).
shared_var(At,C) :- variabili([At],V), variabili([C],W), shared(V,W).
shared([H|_],T) :- membert(H,T).
shared([_|L],T) :- shared(L,T).
/* riblock
   1st arg: the list of blocks to be merged in a new block          input
   2nd arg: the last atom                                           input
   3rd arg: the list of the blocks, in the right order of course    input
   4th arg: the new list of blocks, special care as been taken to   output
            make sure that a block comes before another one if its
            first atom comes before  the first atom of the other
            block in the body of the clause
*/
riblock(L,At,[H|R],[New_block|Rest_blocks]) :-
   member3(L,H,L1),!, make_block(L,At,New_block), filter(L1,R,Rest_blocks).
riblock(L,At,[H|R],[H|Z]) :- riblock(L,At,R,Z).
/* make_block
   1st arg: a list of blocks to merge, the order is important       input
   2nd arg: the last atom of the new block                          input
   3rd arg: the new block was obtained appending in the right order output
            the blocks in 1st and placing 2nd at the end
*/
make_block([],At,[At]).
make_block([H|R],At,Bl) :- make_block(R,At,Bl1), append(H,Bl1,Bl).
/* filter
   1st arg: a list                                                  input
   2nd arg:  a second list                                          input
   3rd arg: the list of the elements of 2nd that aren't             output
            elements of 1st, the order is otherwise kept
*/
filter(_,[],[]).
filter(L,[H|R],Z) :- membert(H,L),!, filter(L,R,Z).
filter(L,[H|R],[H|Z]) :- filter(L,R,Z).
/* distri1
   1st arg: the list of all the atoms in the body of the clause     input
   2nd arg: the list of the blocks of the clause                    input
   3rd arg: the list of the base and irrelevant atoms in the order  input
            in which they occur in the body of the clause
```

```
   4th arg: the list of the atoms not relevant before the first         output
            relevant atom
   5th arg: the list of the subclauses in which I've divided the        output
            clause
   For all lists above mentioned the order is very important, in the
   definition of the procedure I've taken advantage of it in several places
*/
distri1(_,Blxx,[],[],Blxx).
distri1([At1|Rbd],Blxx,[At1|Rirr],[At1|Rh],Mess) :-
                         distri1(Rbd,Blxx,Rirr,Rh,Mess).
distri1([At1|Rbd],Blxx,[At2|Rirr],[],Mess) :- At1 \== At2,
                         distri2(Rbd,Blxx,[At2|Rirr],Mess).
/* distri2
   1st arg: the list of all the atoms of the clause                     input
   2nd arg: the list of the blocks of the clause                        input
   3rd arg: the list of the base and irrelevant atoms                   input
   4th arg: the list of the subclauses in which I've divided the        output
            clause
*/
distri2(_,Blxx,[],Blxx).
distri2([At1|Rbd],[Bl1,Bl2|Rbxx],[At1|Rirr],Mess) :-
                         distri2(Rbd,[[At1|Bl1],Bl2|Rbxx],Rirr,Mess).

/* in the above clause the order of the non relevant atoms is inverted, I'll
 correct this in get_tlock, instead in the last clause I need to invert on
purpose the order so that when I invert again in get_tlock I have at last
the rigth order
*/
distri2([At1|Rbd],[Bl1,Bl2|Rbxx],[At2|Rirr],Mess) :-
   At1 \== At2, membert(At1,Bl1),
   distri2(Rbd,[Bl1,Bl2|Rbxx],[At2|Rirr],Mess).
distri2([At2|Rbd],[Bl1,Bl2|Rbxx],[At1|Rirr],[Bl1|Mess]) :-
   At1 \== At2, not_member(At2,Bl1), membert(At2,Bl2),
   distri2(Rbd,[Bl2|Rbxx],[At1|Rirr],Mess).
distri2([At2|Rbd],[Bl1,Bl2|Rbxx],[At1|Rirr],Mess) :-
   At1 \== At2, not_member(At2,Bl1), not_member(At2,Bl2),
/* At2 is an atom of a previus block */
distri2(Rbd,[Bl1,Bl2|Rbxx],[At1|Rirr],Mess).
distri2(_, [Bl1],[A|Rirr],[Bl2]) :- reverse([A|Rirr],Rirr1),
                                       append(Rirr1,Bl1,Bl2).

/* distri1 distibutes the non relevant atoms in the body of the clause
according to the following guidelines:
1)if the non relevant atom At comes before any relevant atom in the body
of the tclause then place At in the list of atoms before any block, these
atoms will stay with the head of the tclause.
2) if At comes after the first atom of the i-th block and before the first
atom of the (i+1)-th block then  At is added to the i-th block.
distri2 is called when there has already been a relevant atom, so there isn't
any more the business of finding the atoms of the 1st kind.
a) If the atom At1  under examination is a non relevant one it is added to
the present block then we can go on to examine the nest atom of the clause.
```

```
b) If the At1 is not equal to the first non relevant atom of the ordered list
of non relevant atoms then At1 is relevant:
b1) if At1 is part of the present block the present block doesn't change, we
examine the next atom in the clause;
b2) if At1 is not part of the present block Bl1 but instead it's part of the
following block we wont need to add any more non relevant atoms to Bl1,Bl2
becomes the present block, we examine the next atom in the clause;
b3) if At1 isn't part of Bl1 nor of the next block Bl2 I belive At1 must be
part of a past block, just go on;
b4) if there is only one block left then it will get all the non relevant
atoms left.
*/

/* get_tlock
   1st arg: a T-subclause                                             input
   2nd arg: the list of the relevant predicates                       input
   3rd arg: the tlock from the T-subclause                            output
   4th arg: the head of the T-subclause followed by the non relevant  output
            predicates
   5th arg: the link between 4th and 3rd                              output
*/
get_tlock([Head|Tscl],Pred,Tlo,[Head|Bds],Links) :-
  find_non_rel(Tscl,Pred,Bds1,Relev), reverse(Bds1,Bds),
  variabili([Head|Bds],Lv1), variabili(Relev,Lv2),
  links(Lv1,Lv2,Links), copy_term([Links|Relev],Tlo).
/* find_non_rel
   1st arg: the body of the T-subclause                               input
   2nd arg: the list of the relevant atoms                            input
   3rd arg: the list of the non relevant atoms                        output
   4th arg: the list of the relevant atoms                            output
*/
find_non_rel([At1|Rest],Pred,[At1|R_irr],Rel) :-
      functor(At1,Nam,Ar),
      not_member([Nam,Ar],Pred),
      find_non_rel(Rest,Pred,R_irr,Rel).
find_non_rel([At1|Rest],Pred,[],[At1|Rest]) :-
   functor(At1,Nam,Ar), membert([Nam,Ar],Pred).

/* put_together
   1st arg: the list of the tclauses descendants of the tlock         input
   2nd arg: the head of the T-subclause                               input
   3rd arg: the non relevant atoms in the T-subclause, they'll go     input
            at the end
   4th arg: the link between the tlock and the T-subclause            input
   5th arg: the list of the tclauses obtained                         output
*/
put_together([],_,_,_,[]).
put_together([Tcl1|Rtcl],Head,Bdd,Link,[Tcl0|Rtcl0]) :-
   copy_term(Tcl1,[Testa|Corpo]),
   copy_term([Head,Bdd,Link],[Heanw,Bddnw,Lknw]), Lknw = Testa,
   append([Heanw|Corpo],Bddnw,Tcl0), put_together(Rtcl,Head,Bdd,Link,Rtcl0).
```

```
/* links_care
   1st arg: the entire tclause relevant and irrelevant predicates      input
   2nd arg: the list of the bloxx in which 1st has been divided        input
   3rd arg: the list of the tloxx of 1st                               output
   4th arg: the list of the links between each tlox and 1st            output
*/
links_care(_,[],[],[]).
links_care(Tcl,[Bx1|Bxx],[[Lv1|Bx1]|Rtxx],[Lv1|Rlnx]) :-
  linking_vars(Bx1,Tcl,Lv1), links_care(Tcl,Bxx,Rtxx,Rlnx).
/* linking_vars
   1st arg: a list of atoms                                            input
   2nd arg: a second list of atoms including 1st                       input
   3rd arg: the list of the linking variables of 1st in 2nd            output
*/
linking_vars(Bl,[H|R],L):-variabili(Bl,L1),
   remove_predicates(R,Bl,R2), variabili([H|R2],L2), links(L1,L2,L),!.
/* links
   1st arg: a list                                                     input
   2nd arg: a second list                                              input
   3rd arg: the list of the common elements of the two list            output
*/
links([],_,[]).
links([H|R],L,[H|R1]):-membert(H,L), links(R,L,R1).
links([H|R],L,R1):- not_member(H,L), links(R,L,R1).
/* remove_predicates
   1st arg: a list of atoms                                            input
   2nd arg: a second list of atoms                                     input
   3rd arg: the list of the elements of the first list which           output
            aren't elements of the second list
*/
remove_predicates([],_,[]).
remove_predicates([H|R],Bl,R2):-membert(H,Bl),
                                remove_predicates(R,Bl,R2).
remove_predicates([H|R],Bl,[H|R2]):-remove_predicates(R,Bl,R2).
/* permute_furb
   1st arg: a list of atoms                                            input
   2nd arg: a second list of atoms, the length  is =< of 1st           input
   3rd arg: a permutation of 1st, or of a sublist of 1st, if there     output
            is any, such that each element of 3rd is an instance of
            the corresponding element  of 2nd
*/
permute_furb(_,[],[]).
permute_furb(S,[G|K],[M|J]) :- member_match(S,G,S1,M), permute_furb(S1,K,J).
/* member_match
   1st arg: a list of atoms                                            input
   2nd arg: an atom                                                    input
   3rd arg: if in 1st there is an atom, which is an istance of 2nd     output
            the list is returned without this atom
   4th arg: the istance of 2nd                                         output
*/
member_match([S|L],G,L,S) :- subsumes_chk(G,S).
member_match([S|L],G,[S|K],M) :- member_match(L,G,K,M).
```

```
/* faithful_instance
   1st arg: a tlock                                                 input
   2nd arg: a second tlock                                          input
   3rd arg: the permutation of 1st if this is a faithful instance   output
            of 2nd
    The goal succeeds if a permutation of 1st is a faithfull instance
    of 2nd; that is an istance and also the fact that to every existencial
    variable corresponds an existencial variable and no double occurrence
    of variables in the range of the bindings
*/
faithful_instance([Vs|S],[Vg|G],[Vs|Per_S]) :-
  length(S,Ns), length(G,Ng), Ns=Ng, permute_furb(S,G,Per_S),
  find_exist_vars(Per_S,Vs,Ex_var), fi([Vg|G],[],[],Ex_var,[Vs|Per_S]).
/* fi
   1st arg: a block                                                 input
   2nd arg: the list of links between existencial variables         input
            in the two blocks; set to [] on first call
   3rd arg: the list of links between a linking var and a term      input
            without variables occurring more than once; [] first call
   4th arg: the list of the existencial variables of 5th            input
   5th arg: a second block                                          input
   The goal succeeds if 5th is, no permutations allowed, a faithful
   instance of 1st
*/
fi([_],_,_,_,[_]).
fi([La|[A1|A]],E,Lv,Evar,[Lb|[B1|B]]) :-
   ground(A1), ground(B1), A1==B1, fi([La|A],E,Lv,Evar,[Lb|B]).

fi([La|[A1|A]],E,Lv,Evar,[Lb|[B1|B]]) :-
   var(A1), not_member(A1,La), var(B1), membert(B1,Evar), !,
   /* two existencial variables */
   look_in_exis_ls(A1,B1,E,E1), fi([La|A],E1,Lv,Evar,[Lb|B]).

fi([La|[A1|A]],E,Lv,Evar,[Lb|[B1|B]]) :- var(A1), membert(A1,La),
   !, /* the first is a linking variable */
   (ck_term([B1],Lb,Nlb) -> ck_bindings(A1,B1,Lv,Lv1);
   (ck_bindings(A1,B1,Lv,Lv), Lv=Lv1,Lb=Nlb)),  fi([La|A],E,Lv1,Evar,[Nlb|B]).

/* the first ck_binding checks if something wasn't already given to A1, instead
 the second ck_bindings succedes only if a binding for A1, B1 has already been
established */
fi([La|[A1|A]],E,Lv,Evar,[Lb|[B1|B]]) :- compound(A1), compound(B1),
   functor(A1,Nm,Ar), functor(B1,Nm,Ar), A1=..[_|Ma], B1=..[_|Mb],
   append(Ma,A,A2), append(Mb,B,B2), fi([La|A2],E,Lv,Evar,[Lb|B2]).
/* look_in_exis_ls
   1st arg: an existencial variable                                 input
   2nd arg: a second existencial variable                           input
   3rd arg: a list of links between existencial variables           input
   4th arg: if there already exist a link between the two variables     output
            the 3rd list is returned, if none of the two variables
            occurs in any existing link, the link between them is added
*/
```

```
look_in_exis_ls(A1,B1,[],[[A1,B1]]).
look_in_exis_ls(A1,B1,[[A,B]|Rest],[[A,B]|Rest1]) :- A1\==A, B1\==B,
   look_in_exis_ls(A1,B1,Rest,Rest1).
look_in_exis_ls(A1,B1,[[A,B]|Rest],[[A,B]|Rest]) :- A1==A, B1==B .
/* ck_term
   1st arg: a list of terms                                          input
   2nd arg: the list of the linking variables of the block to which     input
            the term belongs, please no repetitions
   3rd arg: the new list of the linking variables. The variables
            which have been found in 1st were removed from 2nd.
   The goal succeeds if all the variables are linking, and they belong to 2nd
*/
ck_term([],S,S).
ck_term([B|R],Lb,Nbl) :- ground(B), ck_term(R,Lb,Nbl).
ck_term([B|R],Lb,Nbl) :- var(B), member3(Lb,B,Lb1), ck_term(R,Lb1,Nbl).
ck_term([B|R],Lb,Nbl) :-
   compound(B), B=..[_|L], append(L,R,R1), ck_term(R1,Lb,Nbl).
/* ck_bindings
   1st arg: a linking variable                                       input
   2nd arg: a term                                                   input
   3rd arg: a list of bindings between variables and terms           input
   4th arg: if a binding between the 1st and 2nd already exists 3rd     output
            is returned, if a binding between 1st an some term
            different from 2nd exist, the goal fails, otherwise
            the new binding is added to the list
*/
ck_bindings(A,B,[],[[A,B]]).
ck_bindings(A1,B1,[[A,B]|L],[[A,B]|L]) :- A==A1, B==B1.
ck_bindings(A1,B1,[[A,B]|L],[[A,B]|L1]) :- A\==A1, ck_bindings(A1,B1,L,L1).
/* member3
   1st arg: a list                                                   input
   2nd arg: a term                                                   input
   3rd arg: if the term belongs to the list, the goal succeds and       output
            the rest of the list is returned in the same order as they
            appear in 1st
*/
member3([A|L],At,L) :- A==At.
member3([A|L],At,[A|L1]) :- A\==At, member3(L,At,L1).
%   predicate just exists in aioni's transformer.
%
/* find_exist_vars
   1st arg: a list of atoms, i.e. a block                            input
   2nd arg: the list of the linking variables of 1st                 input
   3rd arg: the list of the existencial variables of 1st             output
*/
find_exist_vars(Bl,Lv,Ev) :-
   variabili(Bl,Variabili),setminus(Variabili,Lv,Ev).
/* setminus
   1st arg: the list of all the variables of a block                 input
   2nd arg: the list of the linking variables of the block           input
   3rd arg: the list of the existencial variables of the block       output
*/
```

```
setminus([], _, []).
setminus([A|Vrs], Lv, E) :-
   membert(A,Lv) -> setminus(Vrs,Lv,E); (setminus(Vrs,Lv,E1),  E = [A|E1]).
/* gen_step
   1st arg: the block in need of generalization                         input
   2nd arg: the block after the generalization                          input
*/
gen_step([Head|Bl],[Head|Bg]) :-
   bound(H,V), normV(Bl,V,Nbl,L,[]),normH(Nbl,H,Eq,New_bl),
   append(Eq,L,Total_eq), append(New_bl,Total_eq,Bg).
/* normV
   1st arg: list of atoms,                                              input
   2nd arg: natural number,                                             input
   3rd arg: the generalisation of 1st such that the heigth of each      output
            atom in this new list is =< than 2nd, the order is
            preserved.
   4th arg: the list of equalities to add to 5th                         output
   5th arg: the list of equalities to start with, [] on first call       input
*/
normV([], _, [], [],_).
normV([T|L], 0, [V|La], L_out,L_in) :-
   compound(T), (search(T,L_in,V) -> normV(L,0,La,L_out,L_in);
   (copy_term(_,V), normV(L,0,La,L_out1,[T=V|L_in]), [T=V|L_out1] = L_out)).
normV([T|L], N, [T|La], L_out,L_in) :- (var(T); atomic(T)),
    normV(L, N, La, L_out,L_in).
normV([T|L], N, [Ta|La], L_out,L_in) :- N>0, compound(T), T=..[P|Arg],
  M is N-1, normV(Arg,M,AArg,L1,L_in), Ta=..[P|AArg],
  append(L1,L_in,L2), normV(L,N,La,L3,L2),  append(L1,L3,L_out).
/* normH
   1st arg: a block, that is a list of atoms,                           input
   2nd arg: the horizontal bound                                        input
   3rd arg: the list of the equalities  added to split  1st             output
   4th arg: the horizontal generalization of 1st                        output
*/
normH(Bl,H,[],Bl) :- length(Bl,Lh), Lh=<H.
normH(Bl,H,E,Hgl) :- length(Bl,Lh),Lh>H,  bagof(M,spezza(Bl,H,M),Lm),
                     find_best(Lm,[Vc,X]), sostitut(Bl,X,Vc,[],Hgl,E).
/* spezza
   1st arg: a list of atoms, as for example a block,                    input
   2nd arg: natural namber                                              input
   3rd arg: a couple of lists: the second is a subsequence of 1st       output
            first is  the list of the linking variables betwen the
            subsequence  and 1st
*/
spezza(Bl,H,[Vc,X]) :- subseq(X,Bl), length(X,Nx),
   Nx<H, ((X=[At], At=..[_|Args], not_all_diff_vars(Args))  ;
   X =[_,_|_]), filter(X,Bl,Y),
    ((Y=[At3], At3=..[_|Args1], not_all_diff_vars(Args1)) ; Y= [_,_|_]),
     variabili(X,Vx), variabili(Y,Vy), links(Vx,Vy,Vc).
not_all_diff_vars([A|_]) :- nonvar(A).
not_all_diff_vars([A|L]) :-
   var(A), (membert(A,L) ->true; not_all_diff_vars(L)).
```

```
/* variabili
   1st arg: list of terms                                             input
   2nd arg: list of the variables of 1st, no repetitions              output
*/
variabili([],[]).
variabili([H|R],S) :- var(H), variabili(R,R1), app_sine_repet([H],R1,S).
variabili([H|R],R1) :- ground(H), variabili(R,R1).
variabili([H|R],R1) :- compound(H), H=..[_|X],
    variabili(X,L), variabili(R,L1), app_sine_repet(L,L1,R1).
/* subseq
   1st arg: a list                                                     output
   2nd arg: a subsequence of 1st, i.e. the elements of it are          input
            elements of the list and they occur in the same order
            of the list
*/
subseq([A],[A|_]).
subseq([A|X],[A|B]) :- subseq(X,B).
subseq([A|X], [_|B]) :- subseq([A|X],B).
/* sostitut
   1st arg: list of terms\atoms                                        input
   2nd arg: sublist of 1st                                             input
   3rd arg: the list of the linking variables betwen 1st and 2nd       input
   4th arg: list of bindings betwen one of  3rd's variables and        input
            a fresh variable; set at [] on  first call
   5th arg: the 1st changed so that the links betwen 2nd and the       output
            rest of 1st have been severed i.e. there are no more
            linking variables betwen them
   6th arg: the list of equalities that had to be added to 4th         output
            to bring home the bacon, that is 5th
*/
sostitut([A|Bl], X, Vc, L_in, [C|B_out], L_out) :-
  membert(A,X), sost_var([A], Vc, L_in, L, [C]), append(L, L_in, L2),
  sostitut(Bl,X,Vc,L2,B_out,L3), append(L,L3,L_out).

sostitut([A|Bl],X,Vc,L_in,[A|B_out],L_out):-
   not_member(A,X),  sostitut(Bl,X,Vc,L_in,B_out,L_out).
sostitut([],_,_,_,[],[]).
/* sost_var
   1st arg: a list of atoms/terms                                      input
   2nd arg: a list of variables                                        input
   3rd arg: list of bindings betwen one of the 3rd variables and       input
            a fresh variable; set at [] on first call
   4th arg: the list of equalities that had to be added to 4th         output
            to get 5th
   5th arg: the 1st list has been changed so that there are no more    output
            occurences of the 2nd variables, they have been
            replaced according with 1st and 4th
*/
sost_var([],_,_,[],[]).
sost_var([A|R], Vc,L_in,[A=D|L_out],[D|S]) :-
   var(A), membert(A,Vc), search_inv(A,L_in),
   copy_term(_,D), sost_var(R,Vc,[A=D|L_in],L_out,S).
```

```
sost_var([A|R], Vc,L_in,L_out,[A|S]) :-
   var(A), not_member(A,Vc), sost_var(R,Vc,L_in,L_out,S).
sost_var([A|R], Vc,L_in,L_out,[D|S]) :-
   var(A), membert(A,Vc), search(A,L_in,D),
   sost_var(R,Vc,L_in,L_out,S).
sost_var([A|R], Vc,L_in,L_out,[A|S]) :-
   ground(A), sost_var(R,Vc,L_in,L_out,S).
sost_var([A|R], Vc,L_in,L_out,[D|S]) :-
   \+(ground(A)), compound(A), A =..[P|Arg],
   sost_var(Arg,Vc,L_in,L1,AArg), D =..[P|AArg], append(L1,L_in,L2),
   sost_var(R,Vc,L2,L3,S), append(L1,L3,L_out).
/* search_ inv
   1st arg: a variable or a term                                        input
   2nd arg: a list of bindings like variable/term = variable            input
   the goal succeds if there is no binding involving 1st
*/
search_inv(_,[]).
search_inv(A,[D|L]) :- D= (S=_), S\==A, search_inv(A,L).
/* search
   1st arg: a variable/term                                             input
   2nd arg: a list of bindings like "variable/term = variable"          input
   3rd arg: if there exists a binding involving 1st, it reports         input
            the other membert
*/
search(A,[S|_],D) :- S=(F=D), F==A.
search(A,[S|L],D) :- S=(F=_), F\==A, search(A,L,D).
/* app_sine_repet
   1st arg: list without repetitions                                    input
   2nd arg: list without repetitions                                    input
   3rd arg: the concatenation of the two lists without repetions        output
*/
app_sine_repet([],M,M).
app_sine_repet([A|L],M,[A|Lr]) :- not_member(A,M), app_sine_repet(L,M,Lr).
app_sine_repet([A|L],M,Lr) :- membert(A,M), app_sine_repet(L,M,Lr).
/* controllo
   1st arg: the block, although the atoms aren't in the right           input
           order, of whom we are going to check the lenght and the
           height of its atoms
   The goal succeeds if at least one of the bounds are superati
*/
controllo([_|Bl]) :-
   bound(_,V), /* bound is a dynamic predicate*/
   ck_depth(Bl,V).
controllo([_|Bl]) :- length(Bl,M), bound(H,_), M>H.
/* ck_depth
   1st arg: the list of atoms of whom we want to check the depth        input
   2nd arg: the vertical bound                                          input
*/
ck_depth([A|_],V) :- depth(A,N), N>V.
ck_depth([_|R],V) :- ck_depth(R,V).
```

```
/* find_best
   1st arg: a list of elements like the following [Vc,X], where         input
            the second is a subsequence, the first is the list of the
            linking variables
   2nd arg: is  the element with the shortest list of linking           output
            variables
*/
find_best([A],A).
find_best([A|L],Best) :- find_best(L,B1), better(B1,A,Best).
/* better
   1st arg: a list like the following one: [A,B] where A is a list      input
            of variables and B is a list of atoms
   2nd arg: same as 1st                                                 input
   3rd arg: the one of the two who has the shorter list of variables    output
*/
better([A,B],[C,D],[A,B]) :-
   length(A,Ca), length(B,Cb), length(C,Cc), length(D,Cd),
   (Ca<Cc ; (Ca=Cc,Cb>Cd)).
better(_,[C,D],[C,D]).
gensym(Prefix, V) :-
   var(V), atomic(Prefix), (retract(gensym_counter(Prefix, M)) ;   M = 0 ),
   N is M+1, asserta(gensym_counter(Prefix, N)), concat(Prefix, N, V), !.
concat(Pref,N,V) :- name(Pref,L), name(N,L1), append(L,L1,S),  name(V,S).
```

C.14 determ_prog.pl

```
% determ_prog.pl
%
% This file contains the predicates which are used by the strategy determ
% which aims at deriving semi-deterministic programs. It is an implementation
% of the strategy presented in:
%     Pettorossi, A., Proietti, M., Renault, S.:
%     Reducing Nondeterminism while Specializing Logic Programs,
%     24th ACM Symposium on Principles of Programming Languages (POPL'97),
%     La Sorbonne, Paris, France, January 15-17, 1997, p. 414-427.
%
% The strategy is parametric wrt to subsidiary strategies which specify
% when and how to unfold, if and when subsumption must be applied, if and how
% case split must be applied. The user may modify the subsidiary strategies
% according to a set of predefined options proposed in the file
% determ_strat_parameters.pl. This file is consulted at the begining of every
% execution of the strategy, and therefore must be updated before running the
% strategy. In particular, the initial program, the initial definition and the
% modes associated with every predicate occurring in the initial program must
% be specified in this file.

:- dynamic current_new_def_functor/1,  initial_def/3.
:- use_module(library(terms), [ variant/2 ]).
/*-------
determ_strat (or simply determ) is the top level goal which applies the
strategy which transform an initial program into a semi-deterministic
specialized one, as described in:
     Pettorossi, A., Proietti, M., Renault, S.:
     Reducing Nondeterminism while Specializing Logic Programs
     24th ACM Symposium on Principles of Programming Languages (POPL'97)
     La Sorbonne, Paris, France, January 15-17, 1997, p. 414-427.
The initial program, the initial specialized clause, predicate modes and
other parameters are set in the file determ_strat_parameters.
--------*/
determ_strat :-
      erase_environment2,
      consult([determ_strat_parameters]),
      show_strat_parameters,
      show_mode_predicates,
      file_input(Input),
      input(Input),
      init_environment, assert_initial_def(X), determ_def(X), !,
      nl, write('Final program : '), nl, nl, pp,
      retractall(strat(_)), retractall(max_depth(_)),
      retractall(comment_history(_,_)),
      retractall(current_depth(_)), retractall(init_prog(_)),
      retractall(mode(_,_,_)), retractall(initial_def(_,_,_)),
      retractall(init_def(_,_,_)),  retractall(tr_session(_)),
      retractall(current_new_def_functor(_)).
determ_strat :- msg_determ_strat_fails.
```

```
/*---------------
 determ_def(Clauses) tries to derive from Clauses a new program where heads of
 clauses are mutually exclusive, w.r.t some input arguments that are given
 with the initial program or infered at run time, for the new definitions.
 When this goal terminates, it gives a new program equivalent to Clauses.
 It might not terminate, due to a non termination in the unfolding strategy,
 or an introduction of a new definition an infinite number of times.
----------------*/
determ_def([]) :- !.
determ_def(Cl0) :-
    unfolding_steps(Cl0,Cl_0), check_trace_session(unf),
    (strat(apply_subsumption(after_unfolding)) ->
     apply_subsumption_clauses(Cl_0,Cl1_0) ; Cl1_0=Cl_0),
     omit_unit_clauses(Cl1_0,Cl1),
     (strat(apply_case_split(global_analysis)) ->
     apply_case_split_global(Cl1,P1,_) ;
     (strat(apply_case_split(binary)) ->
     apply_case_split_binary(Cl1,P1,_) ; P1=[Cl1])),
     check_trace_session(cs),
     (strat(apply_simplification_eq(separate)) ->
     apply_simplification_eq_on_packets(P1,P2) ; P1=P2),
     (strat(apply_subsumption(after_case_split)) ->
     apply_subsumption_on_packets(P2,P) ; P2=P),
     folding_steps_on_packets(P,NewDefs), check_trace_session(fold),
     increment_depth, determ_defs(NewDefs).
determ_defs([]).
determ_defs([NewDef|NewDefs]) :-
     determ_def(NewDef),!, reinitialize_depth, determ_defs(NewDefs).

omit_unit_clauses([],[]).
omit_unit_clauses([C|Cls],Cls2) :-
     is_a_unit_clause(C), !, omit_unit_clauses(Cls,Cls2).
omit_unit_clauses([C|Cls],[C|Cls2]) :- omit_unit_clauses(Cls,Cls2).

folding_steps_on_packets([Packet|Packets],[NewDef|NewDefs]) :-
    remove_clauses_non_subject_to_folding(Packet,Packet2),
    (Packet2=[] -> NewDef=[] ; folding_one_packet(Packet2,NewDef)),
     folding_steps_on_packets(Packets,NewDefs).
folding_steps_on_packets([],[]).

% Removal of clauses with empty bodies or bodies made of inequalities only.
remove_clauses_non_subject_to_folding([],[]).
remove_clauses_non_subject_to_folding([E|L],L2):-
    clause3(E,(_,[]),_), !, remove_clauses_non_subject_to_folding(L,L2).
remove_clauses_non_subject_to_folding([E|L],L2) :-
    clause3(E,(_,B),_), body_without_inequalities(B,_,[]), !,
    remove_clauses_non_subject_to_folding(L,L2).
remove_clauses_non_subject_to_folding([E|L],[E|L2]) :-
        remove_clauses_non_subject_to_folding(L,L2).
folding_one_packet(Packet,NewDef) :-
    (try_to_fold_with_defs(Packet) -> NewDef=[] ;
    introduce_new_def_and_fold(Packet,NewDef)).
```

```
/*---------------------- FOLDING STRATEGY --------------------------*/
try_to_fold_with_defs(Clauses) :-
   node(Pi,_,_,(gen_def,LDefs), _), length(Clauses,N),
   length(LDefs,N2), N2 >= N, try_to_fold_with_defs(Pi,Clauses,LDefs).

/*-----------
Here we allow only complete folding (the whole set of defs and whole body)
------------*/
try_to_fold_with_defs(Pi,Clauses,Defs):-
   heads_unifiable(Defs,_,A),
   fold_cond2(Clauses,Defs,_,_,NewClause),
   definition_pred_used_for_folding(Defs,A),
   msg_user_successful_folding(Pi,A),
   add_clauses(NewCN,[NewClause],nil), %NewCN=[X],
   update_programs((Clauses,(fold, Defs),NewCN), _),
   new_clauses(fold,Clauses,Defs,NewCN).

complete_folding([C],[D],_,NewClause) :-!,
   clause3(C, (Hd_C,IneqsBd_C),_),
   clause3(D,(Hd_D,Bd_D),_),
   body_without_inequalities(IneqsBd_C,Ineqs,Bd_C),
   copy_term(Bd_C,C_Bd_C), Bd_C=Bd_D,
   variant(Bd_C,C_Bd_C),
   append([Ineqs,[Hd_D]],IneqsHd_D),
   NewClause= (Hd_C, IneqsHd_D).
complete_folding([C|LC], [D|LD], A, New_Clause):-
   complete_folding([C|LC], [D|LD], A, [], [], New_Clause).
complete_folding([], [], _,  L_Rec_A, L, NewClause) :-
   anti_unify_list(L_Rec_A,A), recover2_theta_i(L_Rec_A,A,L_Theta_i),
   recover_H_AFG( L,L_Theta_i,NewClause).
complete_folding([C|LC], LD, A, L_Rec_A, L, NewClause) :-
   clause3(C, (Hd_C,IneqsBd_C),_), del_und(D,_,LD,LD2),
   clause3(D,(Hd_D,Bd_D),_), copy_term(A,Copy_A),
   Copy_A=Hd_D, body_without_inequalities(IneqsBd_C,Ineqs,Bd_C),
   verify_instantiation_body(Bd_C, Bd_D, Hd_D),
   Bd_C=Bd_D, append([Ineqs,[Hd_D]],Y), New= (Hd_C, Y),
   complete_folding(LC, LD2, A, [Hd_D|L_Rec_A], [New|L], NewClause).

introduce_new_def_and_fold(Clauses,NewDefsN) :-
   gen_new_name_functor(X), anti_unify_heads(Clauses,Head,U),
   Head =.. [_|L2], new_defs_general_case(U,X,L2,NewDefs2),
   rearrange_definitions(NewDefs2,NewDefs3),
   try_to_drop_arguments(NewDefs3, NewDefs),
   nl, write('Introduction of definition '), write(X), write(':'),nl,
   apply_gen_define(NewDefs,NewDefsN),
   current_prog(PN,_), assert(initial_def(X,PN,NewDefsN)),
   f(Clauses,NewDefsN), assert_modes_new_def(NewDefsN).

new_defs_general_case([],_,_,[]).
new_defs_general_case([(H,B1)],X,L2,[(NewH,B2)]) :-!,
   new_defs_general_case_body(B1,B2), H =..[_|L1],
   new_defs_general_case_args(L1,L2,L3), NewH =..[X|L3].
```

```
new_defs_general_case([(H,B1)|LL1],X,L2,[(NewH,B2)|LL2]) :-
   new_defs_general_case_body(B1,B2), H =..[_|L1],
   new_defs_general_case_args(L1,L2,L3),
   NewH =..[X|L3], new_defs_general_case(LL1,X,L2,LL2).
new_defs_general_case_args([],[],[]).
new_defs_general_case_args([A1|L1],[A2|L2], [A1|L3]) :-
   var(A2), !, new_defs_general_case_args(L1, L2, L3).
new_defs_general_case_args([_|L1],[fv|L2], L3) :-
   new_defs_general_case_args(L1, L2, L3).
new_defs_general_case_args([A1|L1],[A2|L2], L3) :-
   A1=..[F|LL1],  A2=..[F|LL2],
   append(LL1,L1,NewL1), append(LL2,L2,NewL2),
   new_defs_general_case_args(NewL1, NewL2, L3).
new_defs_general_case_body([],[]).
new_defs_general_case_body([E|B],B2):-
   (functor(E, \+,_);functor(E,=,_)), !,
   new_defs_general_case_body(B,B2).
new_defs_general_case_body([E|B],[E|B2]):-
   new_defs_general_case_body(B,B2).

/*-------------------- UNFOLDING STRATEGY --------------------*/

unfolding_steps(Cls1,Cls) :-
     unfold_leftmost_atom(Cls1,Cls2),    % 1st step always leftmost
     unfolding_after_first_step(Cls2,Cls).

unfold_leftmost_atom(Cls1,Cls) :-
        unfold_leftmost_atom2(Cls1,Cls2), !, append(Cls2,Cls).

unfold_leftmost_atom2([E|Cls1],[NewCNs|Cls2]) :-
     unfold_clause_leftmost_atom2(E,NewCNs),
     unfold_leftmost_atom2(Cls1,Cls2).
unfold_leftmost_atom2([],[]).
unfold_clause_leftmost_atom2(E,[]) :-
    body_of_clause(E,R), strat(delete_clauses_with_failing_body),
    \+ non_failed_body(R), !,
    apply_delete_clause(E).
unfold_clause_leftmost_atom2(E,NewCNs) :-
    suitable_prog_to_unfold(E,PN),
    bagof((Head,Body),  one_unf_i(PN,E,1,(Head,Body)),Clauses),
    add_clauses(NewCNs,Clauses,nil),
    update_programs((E,(uunf_i,PN,1),NewCNs),_),
    new_clauses(unfold_d_p,E,1,NewCNs), !.
unfold_clause_leftmost_atom2(E,_) :-
    msg_failure_on_unfolding(E), nltab,
    (yesno('Do you want to apply clause deletion ?', n) ->
    apply_delete_clause(E) ; msg_derivation_fails(E)).

suitable_prog_to_unfold(E,PN):-
    body_of_clause(E,[B1|_]), predname_and_arity(B1,F,A),
    predname_and_arity(N2,F,A),
    (pred_in_initial_prog(N2) -> PN= 1 ; initial_def(F,PN,_)).
```

```
unfolding_after_first_step([],[]).
% Do not unfold unit-clauses
unfolding_after_first_step([E|L_i],[E|L_f]) :-
    is_a_unit_clause(E),!, unfolding_after_first_step(L_i,L_f).
unfolding_after_first_step([C|L_i], L_f) :-
    unfolding_clause_after_first_step(C,LC), !,
    unfolding_after_first_step(L_i, L), append(LC,L,L_f).

unfolding_clause_after_first_step(C,Cls) :-
    body_of_clause(C,R),  \+ non_failed_body(R), !,
    (strat(delete_clauses_with_failing_body) ->
         (apply_delete_clause(C), Cls =[]) ;
        (msg_failure_on_unfolding(C), nltab,
        yesno('Do you want to apply clause deletion ?', n) ->
       (apply_delete_clause(C), Cls=[]) ;  Cls=[C])).
unfolding_clause_after_first_step(C,[C]) :-
     head_of_clause(C,H), strat(unfold_only_non_instantiated_head),
     \+ non_instantiated_head(H), !.
unfolding_clause_after_first_step(C,LC_f) :-
    strat(after_first_step_unfold_leftmost_det), !,
    (unfold_leftmost_det(C,LC) ->
     unfolding_after_first_step(LC, LC_f) ; LC_f=[C]).
unfolding_clause_after_first_step(C,LC_f) :-
    strat(after_first_step_unfold_leftmost_consumer), !,
    (unfold_leftmost_cons(C,LC) ->
    unfolding_after_first_step(LC, LC_f) ; LC_f=[C]).
unfolding_clause_after_first_step(C,LC_f) :-
    strat(after_first_step_unfold_leftmost_weak_consumer), !,
    (unfold_leftmost_weak_cons(C,LC) ->
     unfolding_after_first_step(LC, LC_f) ; LC_f=[C]).
unfolding_clause_after_first_step(C,[C]).

unfold_leftmost_det(E,NewCNs) :-
    suitable_prog_to_unfold_leftmost_det(E,Pos,PN),
    bagof((Head,Body),  one_unf_i(PN,E,Pos,(Head,Body)),Clauses),
    Clauses=[_],!, add_clauses(NewCNs,Clauses,nil),
    update_programs((E,(uunf_i,PN,Pos),NewCNs),_),
    new_clauses(unfold_d_p,E,Pos,NewCNs).

suitable_prog_to_unfold_leftmost_det(E,Pos,PN) :-
    body_of_clause(E,Bd), leftmost_det_unfolding(Bd,1,Pos,PN).

leftmost_det_unfolding([B1|_],Pos,Pos,PN):-
    predname_and_arity(B1,F,A), predname_and_arity(N2,F,A),
    (pred_in_initial_prog(N2) -> PN= 1 ; initial_def(F,PN,_)).
leftmost_det_unfolding([_|Bd],Pos,Pos2,PN):-
   Posi is Pos + 1, leftmost_det_unfolding(Bd,Posi,Pos2,PN).

apply_subsumption_on_packets([],[]).
apply_subsumption_on_packets([P|Ps],[P2|P2s]) :-
   apply_subsumption_clauses(P,P2), apply_subsumption_on_packets(Ps,Ps2).
```

```
apply_simplification_eq_on_packets([],[]).
apply_simplification_eq_on_packets([P|Ps],[NewP|NewPs]) :-
    apply_simplification_eq_on_packet(P,NewP),!,
    apply_simplification_eq_on_packets(Ps,NewPs).
apply_simplification_eq_on_packet([],[]).
apply_simplification_eq_on_packet([C1|C],C2) :-
    clause_number_to_clause(C1, (H,B)),
    apply_unfold_d_eq_(C1, (H,B), removed(_)), !,
    apply_simplification_eq_on_packet(C,C2).
apply_simplification_eq_on_packet([C1|C],[C3|C2]) :-
    clause_number_to_clause(C1, (H,B)),
    apply_unfold_d_eq_(C1, (H,B), C3),
    apply_simplification_eq_on_packet(C,C2).

unfold_leftmost_cons(E,NewCNs) :-
    suitable_prog_to_unfold_leftmost_consumer(E,Pos,PN),
    (bagof((Head,Body),  one_unf_i(PN,E,Pos,(Head,Body)),Clauses) ->
    (add_clauses(NewCNs,Clauses,nil),
    update_programs((E,(uunf_i,PN,Pos),NewCNs),_),
    new_clauses(unfold_d_p,E,Pos,NewCNs)) ; apply_delete_clause(E)).
unfold_leftmost_weak_cons(E,NewCNs) :-
    suitable_prog_to_unfold_leftmost_weak_consumer(E,Pos,PN),
    (bagof((Head,Body),  one_unf_i(PN,E,Pos,(Head,Body)),Clauses) ->
    (add_clauses(NewCNs,Clauses,nil),
    update_programs((E,(uunf_i,PN,Pos),NewCNs),_),
    new_clauses(unfold_d_p,E,Pos,NewCNs)) ; apply_delete_clause(E)).

suitable_prog_to_unfold_leftmost_consumer(E,Pos,PN) :-
    body_of_clause(E,Bd), leftmost_consumer_unfolding(Bd,1,Pos,PN).
suitable_prog_to_unfold_leftmost_weak_consumer(E,Pos,PN) :-
    body_of_clause(E,Bd), leftmost_weak_consumer_unfolding(Bd,1,Pos,PN).

leftmost_consumer_unfolding([B1|_],Pos,Pos,PN):-
    functor(B1,F,A), \+ F =(\+), mode(F,A,Modes),
    consumer(B1,Modes), functor(N2,F,A),
    (pred_in_initial_prog(N2) -> PN= 1 ; initial_def(F,PN,_)).
leftmost_consumer_unfolding([_|Bd],Pos,Pos2,PN):-
    Posi is Pos + 1,
    leftmost_consumer_unfolding(Bd,Posi,Pos2,PN).

leftmost_weak_consumer_unfolding([B1|_],Pos,Pos,PN):-
    functor(B1,F,A), \+ F =(\+), mode(F,A,Modes),
    weak_consumer(B1,Modes), functor(N2,F,A),
    (pred_in_initial_prog(N2) -> PN= 1 ; initial_def(F,PN,_)).
leftmost_weak_consumer_unfolding([_|Bd],Pos,Pos2,PN):-
    Posi is Pos + 1,
    leftmost_consumer_unfolding(Bd,Posi,Pos2,PN).

consumer(B1,Modes) :- B1 =.. [_|Args], consumer_args(Args,Modes).
weak_consumer(B1,Modes) :- B1 =.. [_|Args], weak_consumer_args(Args,Modes).
```

```
consumer_args([],[]).
consumer_args([A-_|L],[M|L2]) :- !,
    (M== '+' -> nonvar(A) ; true), consumer_args(L,L2).
consumer_args([A|L],[M|L2]) :-
    (M== '+' -> nonvar(A) ; true), consumer_args(L,L2).

weak_consumer_args([A|_],[+|_]) :- nonvar(A).
weak_consumer_args([_|L],[_|L2]) :- weak_consumer_args(L,L2).

non_instanciated_head(Hd):-
    functor(Hd,F,N), functor(Hd2,F,N), variant(Hd,Hd2).
non_instanciated_head2(Hd):-
    functor(Hd,F,N), mode(F,N,Modes),
    Hd=..[_|ArgsF], restrict_arg_to_input(ArgsF,Modes,NewArgsF),
    non_instanciated_head_input_args2(NewArgsF).
non_instanciated_head_input_args2([]).
non_instanciated_head_input_args2([A|L]) :-
    non_instanciated_head_input_args2(L),
    (var(A) ; ( A = [B|C], var(B), var(C))).

get_arity_new_def([(H,_)|_], A) :- functor(H,_,A).

try_to_drop_arguments(Defs, NewDefs) :-
   get_arity_new_def(Defs,Arity),
   try_to_drop_arguments(Defs, Arity,NewDefs).
try_to_drop_arguments(Defs, Arity, NewDefs) :-
   useless_arguments(Defs,Arity,Args), drop_arguments(Defs, Args, NewDefs).

useless_arguments(_,0,[]) :- !.
useless_arguments(Defs,Arity,[Arity|Args]) :-
   is_a_useless_argument(Defs,Arity), !, X is Arity - 1,
   useless_arguments(Defs,X,Args).
useless_arguments(Defs,Arity,Args) :-
   X is Arity - 1, useless_arguments(Defs,X,Args).

is_a_useless_argument([],_).
is_a_useless_argument([D|Defs],Arity) :-
   D = (H,B), H =.. [_|Args],
   del_4(Var,Arity,Args,NewArgs), var(Var),
   free_of_var(Var,(NewArgs,B)), is_a_useless_argument(Defs,Arity).

drop_arguments(Defs, [], Defs).
drop_arguments(Defs, [A|L], NewDefs) :-
   drop_argument(Defs, A, NewDefs1),  % must be in decreasing order
   drop_arguments(NewDefs1, L, NewDefs).
drop_argument([],_,[]).
drop_argument([D|Defs], A, [NewD|NewDefs]) :-
   D = (H,B), H =.. [F|Args], del_4(A,Args,NewArgs),
   NewH =.. [F|NewArgs], NewD = (NewH,B),
   drop_argument(Defs, A, NewDefs).
```

C.15 case_split_for_determ.pl

```
% case_split.pl
%
% This file contains the predicates for the case split rule used in the
% strategy determ. It uses the notion of predicate modes. There are two
% options for the rule: binary case split and case split by global analysis.
% Predicate modes are assumed to be set in the file determ_strat_parameters.pl.
% If there are missing, there are requested to the user via a menu.

:- dynamic modes/3.
:- use_module(library(lists), [ append/3, member/2 ]).
:- use_module(library(terms), [ variant/2, subsumes_chk/2,
                                unify_with_occurs_check/2 ]).

/*----------- CASE SPLIT BY GLOBAL ANALYSIS ------------------*/

% Clauses is a list of clause numbers, NewCNS is a list of list of clause
% numbers (packets), NewCN is the flattening of NewCNS.

apply_case_split_global([], [[]],_).
apply_case_split_global(Clauses,NewCNS,NewCN) :-
    split_clauses_into_packets(Clauses,Packets),
    (Packets=[_,_|_] ->
    (add_clauses_packets(Packets,NewCNS), append(NewCNS,NewCN),
    update_programs((Clauses, case_split, NewCN), _),
    new_clauses(case_split, Clauses, NewCNS))
    ; NewCNS=[Clauses], NewCN=Clauses).

split_clauses_into_packets(Clauses,Packets) :-
    restrict_clauses_to_input_heads(Clauses,I_Heads),
    construct_general_structure_of_packets(I_Heads,G_Struct),
    fill_general_structure_of_packets(Clauses,G_Struct,Struct),
    resulting_packets(Struct,Packets).

add_clauses_packets([],[]).
add_clauses_packets([P|Packets],[NewCN|NewCNS]) :-
    add_clauses(NewCN,P,nil), add_clauses_packets(Packets,NewCNS).

restrict_clauses_to_input_heads([],[]).
restrict_clauses_to_input_heads([C|L],LRH) :-
    head_of_clause(C,H), functor(H,F,N), read_modes(F,N,Modes), H=..[_|ArgsF],
    restrict_arg_to_input(ArgsF,Modes,NewArgsF), RH=.. [F|NewArgsF],
    restrict_clauses_to_input_heads(L,Modes,[RH],LRH).

restrict_clauses_to_input_heads([],_,L,L).
restrict_clauses_to_input_heads([C|L],Modes,L_i, LRH):-
    head_of_clause(C,H), H=..[F|ArgsF],
    restrict_arg_to_input(ArgsF,Modes,NewArgsF), RH=.. [F|NewArgsF],
    ((member(E,L_i), variant(E,RH)) ->
    restrict_clauses_to_input_heads(L,Modes,L_i,LRH)
    ; restrict_clauses_to_input_heads(L,Modes,[RH|L_i],LRH)).
```

```
restrict_arg_to_input([],[],[]).
restrict_arg_to_input([X|ArgsF],['+'|Modes],[X|NewArgsF]) :- !,
    restrict_arg_to_input(ArgsF,Modes,NewArgsF).
restrict_arg_to_input([_|ArgsF],[_|Modes],[_|NewArgsF]) :-
    restrict_arg_to_input(ArgsF,Modes,NewArgsF).

construct_general_structure_of_packets(L,L2):-
    infer_missing_heads(L,LL), construct_general_structure_of_packets(LL,[],L2).
construct_general_structure_of_packets([],_,[]).
construct_general_structure_of_packets([H|Heads], Headsi, [ExtH|ExtHeads]) :-
    append(Heads,Headsi,HeadsT), copy_term(HeadsT,HeadsT_C), copy_term(H,H_C),
    infer_extended_heads(H_C,HeadsT_C, ExtH),
    construct_general_structure_of_packets(Heads, [H|Headsi], ExtHeads).

infer_missing_heads([],[]).
infer_missing_heads([H|Heads], All_Heads) :-
    infer_missing_heads(H, Heads, Heads1),
    infer_missing_heads(Heads,Heads2), append(Heads1,Heads2,All_Heads).

infer_missing_heads(H, [], [H]).
infer_missing_heads(H, [H2|Heads], Heads2) :-
   (subsumes_chk(H,H2) ; subsumes_chk(H2,H)
   ; \+ unify_with_occurs_check(H,H2)),!, infer_missing_heads(H, Heads, Heads2).
infer_missing_heads(H, [H2|Hs], [H2_C|Hs2]) :-
   copy_term(H,H_C), copy_term(H2,H2_C), unify_with_occurs_check(H_C,H2_C),
   infer_missing_heads(H, Hs, Hs2).

infer_extended_heads(H_C,HsT_C, ExtH) :-
    more_general_instances(H_C,HsT_C,L),
    infer_constraints(H_C,L, (H_C,[]), ExtH).

more_general_instances(H,Hs,L):-
    all_instances(H,Hs,L2), more_general_instances2(L2,[],L).

all_instances(_,[],[]).
all_instances(H,[H1|Hs],[H1|L]) :-
    \+ variant(H,H1), subsumes_chk(H,H1), !, all_instances(H,Hs,L).
all_instances(H,[_|Hs],L) :- all_instances(H,Hs,L).

more_general_instances2([],I,I).
more_general_instances2([H|Hs],L,L2):-
   (exist_more_general(H,Hs)
   ; exist_more_general(H,L)), !, more_general_instances2(Hs,L,L2).
more_general_instances2([H|Hs],L,L2):-
   append(L,[H],LL), more_general_instances2(Hs,LL,L2).

exist_more_general(H,Hs) :-
   member(H2,Hs), \+ variant(H,H2), subsumes_chk(H2,H).

update_constraints([],Const,Const) :- !.
update_constraints(Const1,Const, NewConst) :- append(Const1,Const,NewConst).
```

```
infer_constraints(_,[], (HC,I), (HC,I,[])).
infer_constraints(HC,[H|L], (HC,I), ExtH) :-
   functor(HC,F,N), read_modes(F,N,Modes), HC=.. [_|ArgsHC], H=.. [_|ArgsH],
   input_args(ArgsHC, Modes, R_ArgsHC), input_args(ArgsH, Modes, R_ArgsH),
   anti_unify(R_ArgsHC,R_ArgsH,T1,T2,_), recover_constraint(T1,T2,Const),
   update_constraints(I,Const,NewConst),
   infer_constraints(HC, L, (HC,NewConst), ExtH).

input_args([],[],[]).
input_args([X|ArgsF],['+'|Modes],[X|NewArgsF]) :-!,
   input_args(ArgsF,Modes,NewArgsF).
input_args([_|ArgsF],[_|Modes],[NewArgsF]) :- input_args(ArgsF,Modes,NewArgsF).

fill_general_structure_of_packets([],Packets, Packets).
fill_general_structure_of_packets([C|L], Packets_i, Packets) :-
   fill_general_structure_with_clause(C, Packets_i, Packets_i2),
   fill_general_structure_of_packets(L, Packets_i2, Packets).
fill_general_structure_with_clause(_, [], []).
fill_general_structure_with_clause(C,[(H,Const,CC)|Hs],
                                        [(H,Const,[(HC_C,BdC_C)|CC])|P_i2]) :-
   clause_number_to_clause(C,(HC,BdC)),
   subsumes_chk_wrt_input_pos(HC,H),!,
   copy_term((HC,BdC), (HC_C,BdC_Ci)),
   copy_term((H,Const),
   (H_C,Const_C)),
   unify_with_occurs_check(HC_C, H_C),
   append(Const_C,BdC_Ci,BdC_C),
   fill_general_structure_with_clause(C, Hs, P_i2).
fill_general_structure_with_clause(C, [X|Hs], [X|P_i2]) :-
   fill_general_structure_with_clause(C, Hs, P_i2).

subsumes_chk_wrt_input_pos(Head,H1) :-
   functor(Head,F,N),
   read_modes(F,N,Modes),
   Head=.. [_|ArgsHead],
   H1=.. [_|ArgsH1],
   restrict_arg_to_input(ArgsHead, Modes, R_ArgsHead),
   restrict_arg_to_input(ArgsH1, Modes, R_ArgsH1),
   subsumes_chk(R_ArgsHead,R_ArgsH1).

resulting_packets([],[]).
resulting_packets([(_,_,Cls1)|Packets],[Cls1|New_Clauses]) :-
   resulting_packets(Packets,New_Clauses).

recover_constraint([],[],[]).
recover_constraint([(_=_)|T1],[(_=X)|T2], Const) :-
   var(X), singleton_variable(X,T2),!,
   recover_constraint(T1,T2, Const).
recover_constraint([(_=A)|T1],[(_=X)|T2],[\+(A=X)|Const]):-
   recover_constraint(T1,T2, Const).

singleton_variable(X,T) :- free_of_var(X,T).
```

```
/*--------------------- BINARY CASE SPLIT -----------------------*/

% The head of clause Clause_Number2 has an input argument which is more
% general than the corresponding argument in the head of clause Clause_Number.
% Clauses is the result of the split of clause Clause_Number and contains
% two clauses.

apply_case_split_binary([], [[]],_).
apply_case_split_binary(Clauses,NewCNS,NewCN) :-
    split_clauses(Clauses,Split1,Split2), (Split2 \== [] ->
    apply_case_split_binary2(Split1,Split2,NewCNS,NewCN) ;
    apply_case_split_binary1(Clauses,NewCNS,NewCN)).

apply_case_split_binary1(Clauses,NewCNS,NewCls) :-
    anti_unify_heads(Clauses,Head,U),
    (exist_clause_to_case_split(Clauses,Head,U,CA) ->
    apply_case_split_binary1_possible(Clauses,CA,NewCNS,NewCls) ;
    apply_case_split_binary1_impossible(Clauses,NewCNS,NewCls)).

exist_clause_to_case_split([_],_,_,_):- !, fail.
exist_clause_to_case_split(Clauses,Head,U,CA) :-
    exist_clause_to_case_split(Clauses,Head,U,CA,Clauses).
exist_clause_to_case_split([C1|_],Head,[(H1,_)|_],C1,AllClauses) :-
    \+ variant(Head,H1), subsumes_chk_wrt_input_mode(Head,H1),
    verify_suitable_clause(H1,Head,AllClauses), !.
exist_clause_to_case_split([_|Clauses],Head,[_|U],C,AllC) :-
    exist_clause_to_case_split(Clauses,Head,U,C,AllC).

subsumes_chk_wrt_input_mode(Head,H1):-
    functor(Head,F,N),
    (read_modes(F,N,Modes) -> subsumes_chk_wrt_input_mode(Modes,Head,H1)
    ; (nl, write('No modes for '), write(F/N), subsumes_chk(Head,H1))).
subsumes_chk_wrt_input_mode(Modes,Head,H1) :-
    Head=.. [_|ArgsHead], H1=.. [_|ArgsH1],
    subsumes_chk_wrt_input_mode_args(Modes,ArgsHead,ArgsH1).
subsumes_chk_wrt_input_mode_args(['+'|_],[A|_],[A2|_]):-
    \+ variant(A,A2), subsumes_chk(A,A2).
subsumes_chk_wrt_input_mode_args([_|Modes],[_|ArgsHead],[_|ArgsH1]):-
    subsumes_chk_wrt_input_mode_args(Modes,ArgsHead,ArgsH1).

verify_suitable_clause(H,Head,[C|_]) :-
    clause_number_to_clause(C,(H1,B1)), anti_unify(H,H1,U),
    variant(U,Head), subsumes_chk_wrt_input_mode(H1,H),
    copy_term(H,C_H), copy_term((H1,B1),(C_H1,C_B1)),
    unify_wrt_input_mode(C_H,C_H1), true_equalities(C_B1,_), !.
verify_suitable_clause(C1,Head,[_|AllClauses]) :-
   verify_suitable_clause(C1,Head,AllClauses).

case_split_clauses(Cls,C,Cls2):- case_split_clauses(Cls,C,[],Cls2).
case_split_clauses([],_,Cls,Cls).
case_split_clauses([C|Cls],C,Cls_i,Cls2) :- !,
    append(Cls_i,[C],Cls_i2), case_split_clauses(Cls,C,Cls_i2,Cls2).
```

```
case_split_clauses([C|Cls],CA,Cls_i,Cls2):-
    head_of_clause(C,H), head_of_clause(CA,H2),
    ((\+ unify_wrt_input_mode(H,H2) ; variant_wrt_input_mode(H,H2)) ->
    (append(Cls_i,[C],Cls_i2), case_split_clauses(Cls,CA,Cls_i2,Cls2))
    ; (apply_case_split(C,CA,Cls1_bef_simpl),
    check_equalities(Cls1_bef_simpl,Cls1),
    add_clauses(NewCNs,Cls1,nil),
    update_programs((C,(case_split,CA), NewCNs),_),
    new_clauses(case_split,C,CA,NewCNs), append(Cls_i,NewCNs,Cls_i2),
    case_split_clauses(Cls,CA,Cls_i2,Cls2))).

apply_case_split(Clause_Number,Clause_Number2,Clauses) :-
    clause_number_to_clause(Clause_Number,(H,B)),
    head_of_clause(Clause_Number2,H2),
    anti_unify(H,H2, Theta_H, Theta_H2,_),
    recover_case_split_arg(H,Theta_H, Theta_H2,A_H,A_H2),
    case_split_arg((H,B),A_H,A_H2,Clauses).

recover_case_split_arg(H,[V1=A_H|_],[V1=A_H2|_],A_H,A_H2):-
    functor(H,F,N), read_modes(F,N,Modes), H =.. [_|Args],
    var(A_H), nonvar(A_H2), occur_in_input(A_H,Args,Modes), !.
recover_case_split_arg(H,[_|L1],[_|L2],A_H,A_H2):-
    recover_case_split_arg(H,L1,L2,A_H,A_H2).

apply_case_split_binary1_impossible(Clauses,[Clauses],Clauses).

apply_case_split_binary1_possible(Cs,CA,NewCNS,NewCls) :-
   case_split_clauses(Cs,CA,NewCs),
   (strat(apply_subsumption(during_case_split)) ->
   apply_subsumption_clauses(NewCs,Cs2)
   ; Cs2=NewCs), apply_case_split_binary(Cs2,NewCNS,NewCls).

apply_case_split_binary2(Split1,Split2,NewCNS,NewCN) :-
   apply_case_split_binary(Split1,NewCNS1,NewCN1),
   apply_case_split_binary(Split2,NewCNS2,NewCN2),
   append(NewCNS1,NewCNS2,NewCNS), append(NewCN1,NewCN2,NewCN).

split_clauses(L,L1,L2):-
   label_arc(_,(_,(case_split,X),_),_), member(X,L), !,
   split_clauses_after_case_split(L,X,L1,L2).
split_clauses([A|C],L1,L2):- split_clauses(C,[A],[],L1,L2,[A|C]).

split_clauses_after_case_split([],_,[],[]).
split_clauses_after_case_split([X|L],X,[X|L1],L2):- !,
   split_clauses_after_case_split(L,X,L1,L2).
split_clauses_after_case_split([C|L],X,[C|L1],L2):-
    label_arc(_,(_,(case_split,X),[C,_]),_), !,
    split_clauses_after_case_split(L,X,L1,L2).
split_clauses_after_case_split([C|L],X,L1,[C|L2]):-
    label_arc(_,(_,(case_split,X),[_,C]),_), !,
    split_clauses_after_case_split(L,X,L1,L2).
```

```
split_clauses_after_case_split([C|L],X,[C|L1],L2):-
    head_of_clause(C,H), head_of_clause(X,H2), unify_wrt_input_mode(H,H2), !,
    split_clauses_after_case_split(L,X,L1,L2).
split_clauses_after_case_split([C|L],X,L1,[C|L2]):-
    split_clauses_after_case_split(L,X,L1,L2).

split_clauses([C1|L],I1,I2,L1,L2,InitSet):-
    no_common_instance_wrt_input_mode(C1,I1),!, append(I2,[C1],I22),
    split_clauses(L,I1,I22,L1,L2,InitSet).
split_clauses([C1|L],I1,I2,L1,L2,InitSet):-
    no_common_instance_wrt_input_mode(C1,I2),!, append(I1,[C1],I11),
    split_clauses(L,I11,I2,L1,L2,InitSet).
split_clauses([],I1,I2,I1,I2,_):-!.
split_clauses(_,_,_,InitSet,[],InitSet).

no_common_instance_wrt_input_mode(N,L):-
    common_instance_wrt_input_mode(N,L), !, fail.
no_common_instance_wrt_input_mode(_,_).

common_instance_wrt_input_mode(N,L):-
    head_of_clause(N,H1), common_instance_with_head_wrt_input_mode(N,H1,L).

common_instance_with_head_wrt_input_mode(N,H,[N3|L]):-
    (label_arc(_,(_,(case_split,_),[N,N2]))
    ; label_arc(_,(_,(case_split,_),[N2,N]))),
    variant_heads(N3,N2), !, common_instance_with_head_wrt_input_mode(N,H,L).
common_instance_with_head_wrt_input_mode(N,H,[N2|L]):-
    label_arc(_,(_,(case_split,N3),[_,N2])), variant_heads(N3,N), !,
    common_instance_with_head_wrt_input_mode(N,H,L).
common_instance_with_head_wrt_input_mode(N,H,[N3|L]):-
    label_arc(_,(_,(case_split,N2),[_,N])), variant_heads(N2,N3), !,
    common_instance_with_head_wrt_input_mode(N,H,L).
common_instance_with_head_wrt_input_mode(_,H1,[N|_]):-
    head_of_clause(N,H2), \+ \+ unify_wrt_input_mode(H1,H2).
common_instance_with_head_wrt_input_mode(N,H1,[_|L]):-
    common_instance_with_head_wrt_input_mode(N,H1,L).

variant_heads(N2,N3) :-
    head_of_clause(N2,H2), head_of_clause(N3,H3), variant(H2,H3), !.

unify_wrt_input_mode(H1,H2) :-
   functor(H1,F,N),
   (read_modes(F,N,Modes) -> unify_wrt_input_mode(H1,H2,Modes)
   ; \+ \+ unify_with_occurs_check(H1,H2)).
unify_wrt_input_mode(H1,H2,Modes) :-
   H1=..[_|ArgsH1], H2=..[_|ArgsH2],
   unify_wrt_input_mode_args(ArgsH1,ArgsH2,Modes).

unify_wrt_input_mode_args([A|ArgsH1],[A2|ArgsH2],['+'|Modes]) :- !,
    unify_with_occurs_check(A,A2),
    unify_wrt_input_mode_args(ArgsH1,ArgsH2,Modes).
```

```
unify_wrt_input_mode_args([_|ArgsH1],[_|ArgsH2],[_|Modes]) :-
    unify_wrt_input_mode_args(ArgsH1,ArgsH2,Modes).
unify_wrt_input_mode_args([],[],[]).

variant_wrt_input_mode(H1,H2) :-
    functor(H1,F,N), (read_modes(F,N,Ms) -> variant_wrt_input_mode(H1,H2,Ms)
    ; variant(H1,H2) ).
variant_wrt_input_mode(H1,H2,Modes) :-
    H1=..[_|ArgsH1], H2=..[_|ArgsH2],
    variant_wrt_input_mode_args(ArgsH1,ArgsH2,Modes).

variant_wrt_input_mode_args(ArgsH1,ArgsH2,Modes) :-
   find_input_mode_args(ArgsH1,ArgsH2,Modes,S_ArgsH1,S_ArgsH2),
   variant(S_ArgsH1,S_ArgsH2).

find_input_mode_args([A|ArgsH1],[A2|ArgsH2],['+'|M],[A|S_ArgsH1],[A2|S_ArgsH2]):- !,
    find_input_mode_args(ArgsH1,ArgsH2,M,S_ArgsH1,S_ArgsH2).
find_input_mode_args([_|ArgsH1],[_|ArgsH2],[_|Modes],S_ArgsH1,S_ArgsH2) :-
    find_input_mode_args(ArgsH1,ArgsH2,Modes,S_ArgsH1,S_ArgsH2).
find_input_mode_args([],[],[],[],[]).
```

C.16 determ_prog_util.pl

```
% determ_prog_util.pl
%
% This file contains predicates used in determ_prog.pl

:- use_module(library(terms), [ variant/2 ]).

:-dynamic comment_history/2.

erase_environment2 :-
  erase_environment, retractall(max_depth(_)),
  retractall(comment_history(_,_)),
  retractall(current_depth(_)), retractall(initial_def(_,_,_)),
  retractall(init_def(_,_,_)), retractall(current_new_def_functor(_)),
  retractall(strat(_)), retractall(mode(_,_,_)),
  retractall(tr_session(_)), retractall(init_prog(_)).

assert_strat_parameters :-
    strat_parameters(S), nl, write('Strategy parameters:'),nl,nl,
    assert_strat_parameters(S).
assert_strat_modes :-
    strat_modes(S), nl, write('Strategy modes:'),nl,nl,
    assert_strat_modes(S).

/*-------------
trace_session/[0,1]
no_trace_session/[0,1]
      is an option which allow the user to interrupt temporarily
      the transformation process at some certain points in order
      to better follow what is going on and/or to insert
      some comments to be inserted in the history text.
      With no argument, the transformation will stop after the unfolding,
      case split and folding steps.
---------------*/
trace_session :-
  asserta(tr_session(unf)), asserta(tr_session(cs)), asserta(tr_session(fold)).

trace_session(Opt) :- asserta(tr_session(Opt)).
no_trace_session :- retractall(tr_session(_)).
no_trace_session(Opt) :- retractall(tr_session(Opt)).

show_strat_parameters :-
   write('Strategy parameters:'), nl, strat(S), write(S), nl, fail.
show_strat_parameters.

show_mode_predicates :-
    mode(_,_,_), !, write('Strategy modes:'), nl, show_mode_predicates2.
show_mode_predicates.

show_mode_predicates2 :-
    mode(F,N,Mode), write(mode(F,N,Mode)), nl, fail.
```

```
show_mode_predicates2.

assert_initial_def(X) :-
    init_def(Pred,Arity,Initial_clauses),
    assert_mode_initial_def(Initial_clauses),
    nl, write('Initial definition:'), nl,nl,
    apply_gen_define(Initial_clauses,X),
    assert(initial_def(Pred,Arity,X)),
    assert(current_new_def_functor(1)).

increment_depth :-
    retract(current_depth(X)), !, max_depth(Xmax),
    (X = Xmax -> msg_derivation_stopped ; X2 is X + 1,
     assert(current_depth(X2))).
increment_depth :- assert(current_depth(1)).
reinitialize_depth :- retractall(current_depth(_)).

check_trace_session(Opt) :- tr_session(Opt),!, get_user_comment.
check_trace_session(_).

get_user_comment :-
    get_line(X), ( X = [] -> true ;
    (X = [112|_] -> pp, get_user_comment ;
    (X = [100|_] -> defs, get_user_comment ;
    (X = [111|_] -> no_trace_session ;
    (X = [99|_] -> insert_comment_history, get_user_comment ;
    (help_trace_session, get_user_comment)))))).

pred_in_initial_prog(Pred):- clause3(_,(Pred,_),prog).

non_instantiated_head(Hd):-
    functor(Hd,F,N), functor(Hd2,F,N), variant(Hd,Hd2).

definition_pred_used_for_folding([C|_],H) :- clause3(C,(H,_),_).

gen_new_name_functor(NewNumberDef) :-
   current_new_def_functor(N), retract(current_new_def_functor(N)),
   N2 is N + 1, assert(current_new_def_functor(N2)),
   name(N2,NameN2), name(NewNumberDef,[110,101,119|NameN2]).

read_modes(P,N,X) :- clause(mode(P,N,X),true), !.
read_modes(P,N,X) :-
  functor(T,P,N), T =.. [_|X], msg_give_modes(P,N),
  give_modes(X,1), assert(mode(P,N,X)).

infer_modes_initial_def([V|L3],B1,['+'|Modes]):-
   occur_in_input(V,B1), !, infer_modes_initial_def(L3,B1,Modes).
infer_modes_initial_def([_|L3],B1,['-'|M]):- infer_modes_initial_def(L3,B1,M).
infer_modes_initial_def([],_,[]).

infer_modes([V|L3],B1,[_|Modes]):-
   occur_in_input(V,B1), !, infer_modes(L3,B1,Modes).
```

```
infer_modes([_|L3],B1,['-'|Modes]):- infer_modes(L3,B1,Modes).
infer_modes([],_,[]).

complete_input_args([]).
complete_input_args([+|L]) :- !, complete_input_args(L).
complete_input_args([_|L]) :- complete_input_args(L).

assert_modes_new_def([D|NewDefsN]) :-
   infer_modes_new_def(D,X,N,ModeArgs),
   assert_modes_new_def(NewDefsN,X,N,ModeArgs).
assert_modes_new_def([],X,N,ModeArgs) :-
   complete_input_args(ModeArgs), assert(mode(X,N,ModeArgs)).
assert_modes_new_def([D|Defs],X,N,ModeArgs) :-
   infer_modes_new_def(D,X,N,ModeArgs),
   assert_modes_new_def(Defs,X,N,ModeArgs).

infer_modes_new_def(X,F,N,ModeArgs) :-
   clause_number_to_clause(X,(H,B)), predname_and_arity(H,F,N),
   H =..[_|L], infer_modes(L,B,ModeArgs).

anti_unify_list([T],T).
anti_unify_list([T1,T2|L], T) :-
   anti_unify(T1,T2,T3), anti_unify_list([T3|L],T).

all_input_modes(0,[]) :- !.
all_input_modes(N,['+'|Modes]):- N2 is N-1, all_input_modes(N2,Modes).
all_input_modes([]) :- !.
all_input_modes(['+'|Modes]):- all_input_modes(Modes).

assert_mode_initial_def([(Head, Body)|_]) :-
   Head=..[F|Args], infer_modes_initial_def(Args,Body,Modes),
   length(Args,N), assert(mode(F,N,Modes)).

occur_in_input(V,[B|_]):-
   functor(B,F,N), read_modes(F,N,Modes), B =.. [_|Args],
   occur_in_input(V,Args,Modes), !.
occur_in_input(V,[_|Bs]):- occur_in_input(V,Bs).
occur_in_input(V,[A|_],['+'|_]) :-
   free_variables(A,[],[],Vars), belongs(V,Vars).
occur_in_input(V,[_|Args],[_|Modes]) :-  occur_in_input(V,Args,Modes).

anti_unify_heads([N], CH, [(H,B)]):-
   (clause3(N, (H,B),_) ; N= (H,B)),
   copy_term((H,B),(CH,CB)), bind_freevars_in_head(CH,CB), !.

anti_unify_heads([N1,N2], H, [(H1,B1), (H2,B2)] ) :-
   (clause3(N1, (H1,B1),_) ; N1=(H1,B1)),
   (clause3(N2, (H2,B2),_) ; N2=(H2,B2)),
    copy_term((H1,B1),(CH1,CB1)), copy_term((H2,B2),(CH2,CB2)),
    bind_freevars_in_head(CH1,CB1), bind_freevars_in_head(CH2,CB2),
    anti_unify(H1,H2,H), !.
```

```
anti_unify_heads([N1,N2|L], H, [(H1,B1), (H2,B2)|L2] ) :-
   (clause3(N1, (H1,B1),_) ; N1=(H1,B1)),
   (clause3(N2, (H2,B2),_) ; N2=(H2,B2)),
    copy_term((H1,B1),(CH1,CB1)), copy_term((H2,B2),(CH2,CB2)),
    bind_freevars_in_head(CH1,CB1), bind_freevars_in_head(CH2,CB2),
    anti_unify(CH1,CH2,H3), anti_unify_heads(L, H4, L2), anti_unify(H3,H4,H).

bind_freevars_in_head(H,IneqsB) :-
    body_without_inequalities(IneqsB,_,B), free_variables(B,[],[],VarsB),
    free_variables(H,VarsB,[],FreeVarsH), bind_vars(FreeVarsH).

bind_vars([]).
bind_vars([fv|L]):- bind_vars(L).
my_concat_atom(L,N) :- name_list(L,Names), append(Names,Name), name(N,Name).

name_list([],[]).
name_list([N|L],[Name|L2]) :- name(N,Name), name_list(L,L2).

del_4(X,1,[X|L],L).
del_4(X,N,[Y|L],[Y|L1]) :- N > 1, M is N-1, del_4(X,M,L,L1).
del_4(1,[_|L],L).
del_4(N,[Y|L],[Y|L1]) :- N > 1, M is N-1, del_4(M,L,L1).
```

C.17 determ_strat_parameters.pl

C.17.1 determ_strat_parameters.pl used in the example of Section 7.3.1

```
:- dynamic max_depth/1, tr_session/1, init_def/3, mode/3, file_input/1, strat/1.

% tr_session allows the user to specify some positions in the transformation
% process where he wants the transformation to stop, until he hits the <CR>
% key.
% Possible positions are after the unfolding steps (unf), case split steps
% (cs), and folding steps (fold).
% One may comment it and/or uncomment the following lines
% according to one's preferences.

%tr_session(unf).  % stops after the unfolding steps
%tr_session(cs). % stops after the case-split steps
%tr_session(fold). % stops after the folding steps

% max_depth sets the depth at which the derivation must stop even if no
% semi-deterministic program has been derived. The depth value corresponds
% to the number of iteration of the loop strategy.
% The following line sets by default the value of this depth variable to 10.

max_depth(10).

% file_input specifies the file which contains the initial program.
% init_def specifies the definition of the clause(s) from which the
% transformation begins.
% init_def has 3 arguments: the first one is the name of the predicate
% in the head of the clause(s) (usually new1), the second one is its arity,
% and the third one is the list of clause(s) that define new1.
% A clause is represented as a pair (Head, Body) where Head is an atom and
% Body a list of atoms. The following two lines specify that the initial
% program is match and the initial definition is the clause
% new1(A) :- match([a,a,b],A), which corresponds
% to the specialization of the predicate match for the input pattern aab.
file_input('Progs/match').
init_def(new1,1,[(new1(A), [match([a,a,b],A)])]).

% strat specifies the value of some parameters for the subsidiary strategies.
% The possible arguments of strat are the following:
% * apply_case_split(global_analysis): case split is applied, according to the
%   global analysis strategy (the whole set of packets is defined in one
%   transformation step only).
% * apply_case_split(binary): binary case split is applied, that is case split
%   is applied as long as there exist two clauses which are not mutually
%   exclusive neither belong to the same packet (that is, have equal heads).
% If none of the above two parameters are selected, the case split rule is
% never applied.
% * apply_subsumption(during_case_split): subsumption is applied after each
%   application of the case split rule. It is meaningful only if the the binary
%   case  split has been selected (since case split by global analysis
%   corresponds to one transformation step only).
```

```
% * apply_subsumption(after_case_split): subsumption is applied after all
%   applications of the case split rule (that is, before folding).
% * apply_simplification_eq(separate)
% * after_first_step_unfold_leftmost_det: after the first unfolding step of a
%   new definition clause (which is always done wrt to the leftmost atom of
%   its body), we go on unfolding each of the resulting clauses wrt the leftmost
%   atom of their body which corresponds to a deterministic unfolding (if it
%   exists).
% * after_first_step_unfold_leftmost_consumer: after the first unfolding step
%   of a new definition clause, we go on unfolding each of the resulting clauses
%   wrt the leftmost atom of their body which is a consumer (if it exists),
%   that is, whose input arguments are all instantiated to non variable terms.
% * after_first_step_unfold_leftmost_weak_consumer: same as above, except that
%   the condition of consumer is relaxed to the one of weak consumer, that is,
%   it is enough that at least one of the input argument is instantiated to a
%   non variable term.
% * unfold_only_non_instantiated_head: after the first unfolding step
%   of a new definition clause, we go on unfolding each of the resulting clauses
%   wrt the leftmost atom of their body, if their heads are non instanciated,
%   that is, are of the form p(X1,...,Xn) where X1,...,Xn are dinstinct variables.
% * delete_clauses_with_failing_body: a clause whose body contains an atom which
%   unify with no head of clause is deleted.
%
% The four lines below specify that:
% 1) the case split rule must be applied according to the 'global analysis'
% method,
% 2) subsumption must be applied after unfolding (that is, before case split),
% 3) after the first unfolding step, we go on unfolding each of the resulting
% clause wrt the leftmost consumer (if possible).
% 4) clauses with a failing body body are deleted.
strat(apply_case_split(global_analysis)).
strat(apply_subsumption(after_unfolding)).
strat(after_first_step_unfold_leftmost_consumer).
strat(delete_clauses_with_failing_body).

% mode(P,N,L) represents the modes associated to predicate P of arity N. L
% is a list of length N, with elements in {+,-}. The i-th element of L
% corresponds to the i-th argument of P. '+' specifies an input argument, that
% is, an argument which should contain a ground term at runtime.
% For the program match, we can choose the following modes:
mode(match,2,[+,+]).
mode(app,3,[-,-,+]).
```

C.17.2 determ_strat_parameters.pl used in the example of Section 7.3.3

```
:- dynamic max_depth/1, tr_session/1, init_def/3, mode/3, file_input/1, strat/1.

% no trace

% max_depth
max_depth(10).

% file_input
file_input('Progs/multi_match').

% init_def
init_def(new1,2,[(new1(A,B), [m_match([[a,a,a],[a,a,b]],A,B)])]).

%strat
strat(apply_case_split(binary)).
strat(delete_clauses_with_failing_body).
strat(after_first_step_unfold_leftmost_weak_consumer).

%modes
mode(m_match,3,[+,+,-]).
mode(match,3,[+,+,-]).
mode(app,3,[-,-,+]).
mode(len,2,[+,-]).
```

References

[1] O. Aioni. A System for the Automatic Transformation of Logic Programs. Technical Report, University of Rome Tor Vergata, January 1995.

[2] J.J. Brunekreef. A Transformation Tool for Pure Prolog Programs. Report P9607, University of Amsterdam, July 1996.

[3] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. Journal of the ACM, 24(1):44–67, January 1977.

[4] M. S. Feather. ZAP Program Transformation System. D.A.I. Reserach Report 54, University of Edinburgh, 1978.

[5] M. Gaspari. Progetto di Interfaccia grafica per l'uso di un trasformatore di programmi logici. Internal Report, University of Rome Tor Vergata, March 1994.

[6] J. W. Lloyd. Foundations of Logic Programs. Springer Verlag, Second edition, 1987.

[7] T. Morgante. Automazione di strategie per la trasformazione di programmi logici. Technical Report, University of Rome Tor Vergata, 1994.

[8] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. Journal of Logic Programming, 19,20:261–320, 1994.

[9] A. Pettorossi, M. Proietti, S. Renault. Enhancing partial deduction via unfold/fold rules. in the Proceedings of the Sixth International Workshop on Logic Program Synthesis and Transformation, LNCS 1207, Springer, 1996.

[10] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. Journal of Logic Programming, 16(1–2):123–161, 1993.

[11] M. Proietti, A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. Theoretical Computer Science, 142(1):89–124, 1995.

[12] A. Pettorossi, M. Proietti and S. Renault. Reducing Nondeterminism while Specializing Logic Programs, in the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL'97) La Sorbonne, Paris, France, pages 414-427, ACM Press, January 15-17, 1997.

[13] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, Proceedings of the Second International Conference on Logic Programming, Uppsala, Sweden, pages 127–138. Uppsala University, 1984.