THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Implementation of loop checking

Henrard, Jean

Award date:
1991

Awarding institution:
University of Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

IMPLEMENTATION OF LOOP CHECKING

by Jean HENRARD

Promoteur :

Professeur B. Le Charlier

Mémoire présenté en vue
de l'obtention du titre
de Licencié et Maître
en informatique

Année académique 1990 - 1991

Acknowledgement

This work has been done during my stay at the Centre for Mathematics and Computer Science (CWI) at Amsterdam.

I thank the staff of CWI for their warm welcome and especially Doctor J.-M. Jacquet and Mr R. Bol for their continuous help and Doctor K. Apt for his helpful guidance. I thank my promotor, Professor B. Le Charlier, for his pertinent remarks about this work.

I would like also to thank Ronald de Groot and Paul Smit for making my stay in Amsterdam so pleasant and the Rotary Club for their financial help.

Résumé

Après une description des mécanismes de loop checking décrits dans [BAK 89] et [B 90], deux implémentations de ces mécanismes sont présentées. La première consiste en un méta-interprèteur, écrit en PROLOG, pour des programmes logiques. La seconde est un pré-compilateur qui transforme un programme PROLOG en un nouveau qui inclu les mécanismes de loop checking.

Une étude comparative de ces différentes implémentations est effectuée, ainsi que la comparaison des différents loop checking. Finalement, la question "Quel est le coût du loop checking?" est posée.

Abstract

After a description of the loop checking mechanisms described in [BAK 89] and [B 90], two implementations of these mechanisms are presented. The first one is a meta-interpreter for logic programs written in PROLOG. The second one is a pre-compiler that transform a PROLOG program into a new one that include the loop checking mechanisms.

A comparison of the different implementations is done, as the comparison of the different loop checking. Finally, the question "How costly is loop checking?" is discussed.

Contents

Chapter 1

Introduction

1.1 History

Since the beginning of computer science, people have tried to make languages that are easier for humans to express themselves in. Starting from assembly language, through FORTRAN, COBOL, PASCAL, C, they all carried the imprint of the underlying machine, named the Von Neumann architecture. Logic has been used as a tool for reasoning about computers and programs, but the use of logic directly as a programming language is quite recent. This has resulted in so-called *logic programming*.

Logic programming is derived from an abstract model, which has no direct dependency on one machine or another. It suggests that the knowledge about the problem and assumptions are sufficient to solve it. A program is a set of axioms and its execution is formalized as the attempt to prove a logical statement.

In 1965, J.A. Robinson published the unification algorithm and the resolution principle [R 65]. Shortly after R. Kowalski formulated the procedural interpretation of Horn clause logic. An axiom

$$A \text{ if } B_1 \text{ and } B_2 \text{ and } \ldots \text{ and } B_n$$

can be read and executed as a procedure of a recursive programming language. In the early 1970's, A. Colmerauer and his group developed a new language, called PROLOG (for PROgrammation en LOGique), based on Kowalski's procedural interpretation.

In the late 1970's D.H.D. Warren developed an efficient implementation of PROLOG. For that purpose he created an abstract machine called the *WAM* (Warren Abstract Machine) [W 84] on which almost all efficient PROLOG implementations are based. It shows that logic programming is a powerful, productive and practical programming formalism.

In the remainder of this chapter, I present logic programming and PROLOG framework. First I give the basic concepts of logic programming (syntax, substitutions, unifications), then I present SLD-resolution and PROLOG itself. Finally I introduce the problem of termination.

1.2 The basic concept

1.2.1 Terminology

First, we suppose the existence of the following sets of symbols:

- set of *variables*;
- set of *functions*. Functions are essentially defined by two components: a name, called the *functor* and an *arity*, which determines the number of arguments the function can take. Function of arity 0 are called *constants*;
- set of *predicates*. Predicates are essentially defined by two components: a name and an arity.

Terms are defined inductively as follows:

- a variable is a term;
- a constant is a term;
- if f is a n-ary function and if $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term. f is called the *functor* of this terms.

If p is an n-ary predicate and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n)$ is an *atomic formula* or just an *atom*. An atom is also called a *positive literal* and its negation $(\neg p(t_1, \ldots, t_n))$ a *negative literal*.

A formula of the form

$$L_1 \vee \ldots \vee L_m$$

where $L_1, \ldots, L_m$ are literals, is called a *clause*. Clauses are implicitly universally quantified. A *Horn clause* is a clause where exactly one literal is positive. It is denoted by

$$H \leftarrow B_1, \ldots, B_m. \quad (\text{if } m > 0)$$

or

$$H. \quad (\text{if } m = 0)$$

with the convention $H \leftarrow B_1, \ldots, B_m$. is a shorthand for $H \vee \neg B_1 \vee \cdots \vee \neg B_m$. The atom H is called the *head* of the clause and the sequence $B_1, \ldots B_m$ is called the *body* of the clause. A *goal* is a clause with only negative literals, it is denoted by $\leftarrow B_1, \ldots, B_m$.

A *procedure* is a finite set of Horn clauses with the same head name and arity. A *Horn clause program* is a set of procedures.

1.2.2 Substitutions and renaming

In logic programming, variables are bound to values by means of substitutions. A *substitution* θ is a finite set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$ where each x_i is a variable, the variables $x_1, \ldots, x_n$ are distinct, and each t_i is a term different from x_i. The substitution is to be read as the variables

$x_1, \ldots, x_n$ are bound to $t_1, \ldots, t_n$ respectively. The empty substitution is represented by ϵ. A pair x_i/t_i is called a *binding*. $\{x_1, \ldots, x_n\}$ is called the *domain* of $\{x_1/t_1, \ldots, x_n/t_n\}$.

If $\{t_1, \ldots, t_n\} = \{x_1, \ldots, x_n\}$ then θ is called a *renaming*.

Substitutions are applied on terms, sequences of literals, clauses. For an expression E and a substitution θ, the application of θ to E, denoted $E\theta$, is the new expression obtained from E by simultaneously replacing each occurrence in E of a variable (x_i) from the domain of θ by the corresponding term (t_i). The expression $E\theta$ is called an *instance* of E. When θ is a renaming substitution then $E\theta$ is called a *variant* of E.

For two substitutions,

$$\theta = \{x_1/t_1, \ldots, x_n/t_n\}$$

and

$$\gamma = \{y_1/s_1, \ldots, y_m/s_m\},$$

the *composition* of θ and γ, denoted $\theta\gamma$, is the set

$$\{x_1/t_1\gamma, \ldots, x_n/t_n\gamma, y_1/s_1, \ldots, y_m/s_m\}$$

where the pairs $x_i/t_i\gamma$ for which $x_i = t_i\gamma$ and the pairs y_i/s_i for which $y_i \in \{x_1, \ldots, x_n\}$ are removed.

A substitution θ is *more general than* a substitution θ' ($\theta \leq \theta'$) if for some substitution γ we have $\theta\gamma = \theta'$.

1.2.3 Unification

A substitution θ is called an *unifier* of two expressions E and F iff it verifies $E\theta = F\theta$. It is called a *most general unifier* (mgu) of E and F if it is more general than any other unifier of E and F.

The following algorithm is based upon Herbrand's original algorithm which deals with solutions of finite sets of term equations. This algorithm was presented in [MM 82].

The standard unification problem can be written as an equation

$$t' = t''.$$

A solution of the equation, a unifier, is a substitution θ, if it exists, which makes the two terms identical.

For the set of equations

$$t'_j = t''_j \quad j = 1, \ldots, k$$

a unifier is a substitution which is simultaneously the solution of all the equations $t'_j = t''_j$.

A set of equations is said to be in *solve form* iff it satisfies the following conditions :

1. the equations are $x_j = t_j$ ($j = 1, \ldots k$) where the x_j's are variables;

2. every variable which is the left member of some equation occurs only there.

A set of equations in solved form has the obvious unifier $\theta = \{x_1/t_1, \ldots, x_k/t_k\}$ and it is its mgu.

Thus to find the mgu of a set of equations is suffices to transformed it into an equivalent set of equations in solved form.

The following algorithm does it if this is possible and otherwise halt with failure.

Given a set of equations, repeatedly perform any of the following transformations. If no transformation applies, stop with failure.

1. $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$
   then replace by the equations $s_1 = t_1, \ldots, s_n = t_n$

2. $f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ where $f \not\equiv g$ or $n \neq m$
   then halt with failure

3. $X = X$ (X is a variable)
   then delete the equation

4. $t = X$ where X is a variable and t is not a variable
   then replace by the equation $X = t$

5. $X = t$ where t is not the same variable as X and X has another occurrence in the set of equations
   if X appears in t (this test is called the *occur check*)
   then halt with failure
   otherwise perform the substitution $\{X/t\}$ on both sides of every other equation

The algorithm terminates when no step can be performed or when failure arises.

1.3 Resolution

A problem is presented to a Horn clause program in the form of a goal, $G = \leftarrow A_1, \ldots, A_n$, called a *goal*. The resolution of the goal G consists of an attempt to prove the following sentence

$$\exists X_1, \ldots, X_l : A_1 \wedge \cdots \wedge A_n$$

with the program clauses considered as axioms of a certain theory, where $X_1, \ldots, X_l$ denote all the variables that appear in $A_1, \ldots, A_n$.

Let P be a Horn clause program and $G = \leftarrow A_1, \ldots, A_n$ a goal. Suppose that $C = A \leftarrow B_1, \ldots, B_m$ is a clause in P such that A_i ($1 \leq i \leq n$), called the *selected* atom, unifies with A with a mgu θ. Then

$$G' = \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots A_n)\theta$$

is called the *resolvent* of G and C with the mgu θ and it is denoted by $G \Rightarrow_{C,\theta} G'$. The rule that chose the selected atom is called the *selection rule*.

An *SLD-derivation* (SLD stands for Selection rule driven Linear resolution for Definite clauses) of $P \cup \{G\}$ is a maximal sequence $G_0, G_1, \ldots$ of goals where $G = G_0$ together with a sequence $C_0, C_1, \ldots$ of variants of clauses from P and a sequence $\theta_0, \theta_1, \ldots$ of substitutions such that for all $i = 0, 1, \ldots$

1. $G_i \Rightarrow_{C_i,\theta_i} G_{i+1}$
2. C_i does not have a variable in common with the derivation up to G_i. This condition is called the *standardization apart*.

The sequence $G_0, G_1, \ldots$ is said *maximal* if it is infinite or the last goal is the empty goal or has no resolvent.

SLD-derivation may be finite or infinite. A finite SLD-derivation may be successful or fail. A successful SLD-derivation is one that ends with the empty goal, denoted by $\square$. And the output is the value assigned to the variables of the query; called the *computed answer substitution* for $P \cup \{G\}$. If $\theta_1, \ldots, \theta_m$ is the sequence of substitutions performed in the derivation to find the empty goal then the computed answer substitution is the restriction of $\theta_1 \ldots \theta_m$ to the variables of the query. A failed SLD-derivation is one that ends with a goal for which it is not possible to find a resolvent w.r.t. P.

For a goal $G = \leftarrow A_1, \ldots, A_n$ it is possible to have more than one clause's head that unifies with A_i, so we can have more than one resolvent for G. An *SLD-tree* as in figure 1.1 is used to represent this; each node represents a goal, the empty goal is represented by $\square$ with the computed answer substitution; if for a goal there is no resolvent then it is surrounded by a box. An arc represents one step in the derivation and is marked with the variant of the clause and the mgu used to find the resolvent.

1.4 PROLOG

PROLOG is certainly the most well-known example of logic programming language.

The variables are strings of letters and digits, beginning with a capital letter; function and predicate symbols are strings of letters and digits beginning with a lower case letter.

In PROLOG program the *leftmost selection rule* is used to find the resolvent. And a PROLOG program is not just a set of clauses but there is an order in the clauses, the order is the order in the text of the program. So the SLD-tree is construct as follow: for a program P and a goal $G = \leftarrow A_1, \ldots, A_n$, to find the resolvent in PROLOG we take the next clause in P, $C = A \leftarrow B_1, \ldots, B_m$, such that A_1 and A unify to find the resolvent of G. And to search in this SLD-tree PROLOG used a depth-first left-to-right search strategy.

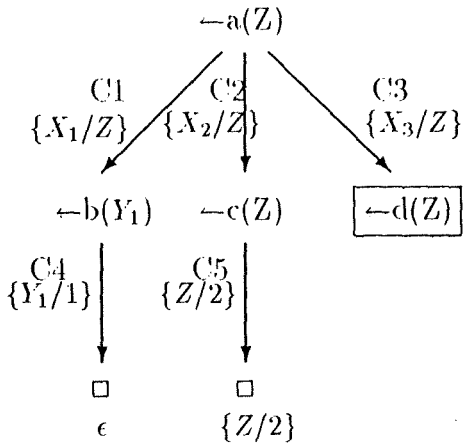

Figure 1.1: The SLD-tree of $P \cup \{\leftarrow \mathsf{a(Z)}\}$
with P = { C1 : a(X) ← b(Y).
C2 : a(X) ← c(X).
C3 : a(X) ← d(X).
C4 : b(1).
C5 : c(2). }

In PROLOG, negation is defined as *negation as failure*, which means that not(G) is "true" if it is not possible to derive G from the program.

PROLOG programs may contain *cuts* (!) which dynamically prune the search tree. For example if we have the following program:

```
P = { C1 : a(X) ← b(X).        C5 : c(4).
      C2 : a(X) ← c(X).        C6 : e(2).
      C3 : b(1).               C7 : e(3).
      C4 : b(X) ← e(X).                  }
```

then the SLD-tree of $P \cup \{\leftarrow a(X)\}$ is represented in figure 1.2. If C3 is replaced by the clause C3' : b(1) ← !. then SLD-tree is the same as in figure 1.2 except the surrounded part of the tree is not developed.

PROLOG has also extra-logical predicates, that have nothing to do with logic programming. They are mainly the I/O predicates, all programming language need them to read and write files, print message on the screen, etc. There are also other extra-logical predicates like assert, retract, bagof.

1.5 Termination

Because of its depth-first search strategy, PROLOG suffers from the problem that it can fall in the exploration of an infinite branch although solutions might be present in some other branches. The undesired consequence is that no solution can be reported although some exists and could be find by other strategies.

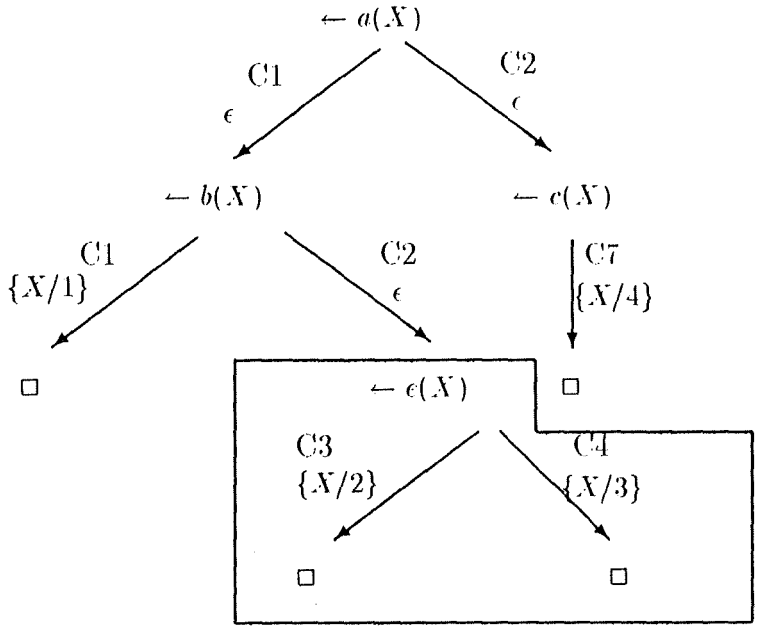

Figure 1.2: The SLD-tree prunes by "!"

Example 1.1 *To illustrate this point, let us consider the transitive closure problem. It consists of finding every path between two nodes of a graph. The relation* r(a,b) *holds iff there is an arc between* a *and* b*. The relation* tc(a,b) *means that there exists a path between* a *and* b*. Let P be the naïve logic program defining the transitive closure* tc *for the relation* r *(given by the facts* F1 – F4*):*

P = { *C1* : tc(X,Z) ← r(X,Z).
C2 : tc(X,Z) ← r(X,Y), tc(Y,Z).
F1 : r(a,b).
F2 : r(b,a).
F3 : r(b,c).
F4 : r(c,d). }.

In the SLD-tree of $P \cup \{\leftarrow tc(a,d)\}$ *(see figure 1.3) there is an infinite branch and the only solution is on the right of this infinite derivation, so PROLOG never reaches that solution.*

This non-termination problem can be solved in two ways:

- transform the program into a new one that does not have infinite branches in its search space;
- change the interpreter, so that it does not go into the infinite branches.

The thesis take place in the second approach, a depth-first search mechanism is used in conjunction with a capability of pruning. Pruning an SLD-tree means that at some point the interpreter is forced to stops its search through a certain part of the tree, typically an infinite branch. Every method of pruning SLD-trees considered so far has been based on excluding some

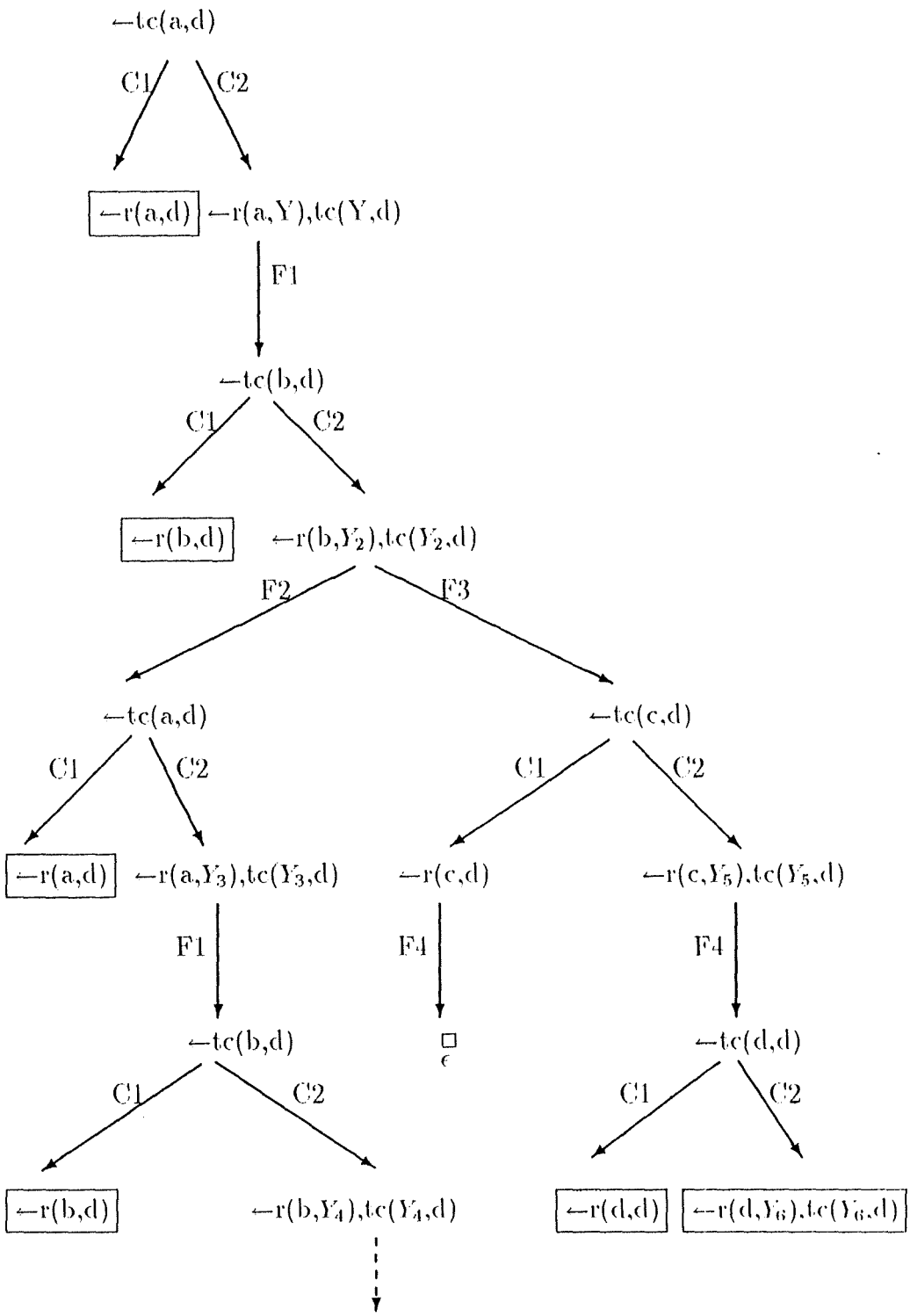

Figure 1.3: SLD-tree of $P \cup \{\leftarrow tc(a,d)\}$

kind of repetition in the SLD-derivations, because such a repetition makes the interpreter enter an infinite loop. That is why pruning SLD-trees has been called *loop checking*.

1.6 Plan of the thesis

Following [BAK 89] and [B 90], in chapter 2 we recall some relevant aspects of the theory of loop checking, together with the definition of the loop checks that are supported in implementation.

In chapter 3 meta-interpreters are described in general. Then the meta-interpreter that supports loop checking is presented. Every part of the computation process, deriving new goals as well as loop checking, is programmed explicitly in PROLOG. This means that the program is slow, but allows to measure the time spent on loop checking separately from the time spent to find the next goal. Finally the output produced by the meta-interpreter are explained.

In chapter 4 a more efficient implementation is presented: the program given by the user is transformed into a new program that includes loop checking (the program is *pre-compiled*). Then the additional layer of intrepretation is removed. First the transformation of program is presented and the pre-compiler with loop checking is described. The search for applicable clauses and unification is done by the underlying PROLOG system, but the computations performed for the loop checks are still explicit. This inequality between the efficiency of generating the next goal and loop checking makes this implementation less suitable for judging the relative cost of loop checking. Some modifications of the pre-compiler are presented to increase the efficiency of the transformed program.

In chapter 5 results obtained by the implementations presented in chapter 3 and 4 are analyse to compare the performance of those implementations. Afterward the loop checks are compared for one of the meta-interpreter, to show the relative cost of the different loop checks. Finally the question "How costly is loop checking?" is discussed, as some possible optimizations of the meta-interpreter.

In the appendix, the PROLOG code of the meta-interpreter and the pre-compiler are given.

Chapter 2

Loop checking

This chapter presents several loop checking mechanisms fully introduced in [BAK 89].

The purpose of loop checking is to prune (stop the search of) infinite branches of the SLD-tree without losing any solution. The loop check prunes a branch when the current goal is "sufficiently similar" to one goal that appears previously in the derivation.

Different loop checks arises from giving different meanings to the term "sufficiently similar".

The ideal loop check may be characterized as follow. Firstly, it should give all the solutions and prune every infinite branch as soon as possible. Furthermore, if there are two branches giving the same solution it only keeps the shortest one. Finally, it should not decrease the efficiency of the interpreter (in time and memory space) for programs without infinite loops. Obviously, such an ideal loop check does not exist and this for at least the two following reasons:

- PROLOG has the full power of recursion theory and therefore the termination of PROLOG programs is undecidable;
- a loop check necessarily keeps some information about the previous goals and compares that with the current goal, so it needs both extra space and time.

We will only be concerned here with loop checks that are computable and independent of the program. They are called *simple loop checks*. Of course simple loop checks that prune every infinite branch and do not lose solutions exist only for some classes of programs.

2.1 General considerations about loop checking

A loop check can be seen as a function from SLD-trees to SLD-trees (subtree of SLD-tree), transforming an SLD-tree to a subtree of it that contains its root and preferably no more the infinite branches. If a node is pruned then it is treated as if failed: all its descendants are removed. In the loop checks

used in this thesis, the decision that a node is pruned depends only on its ancestors in the SLD-tree and not on the program. Because they are based on excluding some repetition in the SLD-derivation (if the pruned node is the same as one of its ancestors).

Obviously, we would like the loop checks to lose no solutions. This corresponds to the concept of soundness. Different degrees of soundness are defined hereafter.

Definition 2.1 (Soundness)

- *A loop check is* **weakly sound** *iff for every program P and goal G, if the SLD-tree of $P \cup \{G\}$ contains a successful branch with a computed answer substitution θ then the pruned SLD-tree of $P \cup \{G\}$ contains also a successful branch with a computed answer substitution θ' possibly different from θ.*

- *A loop check is* **sound** *iff for every program P and goal G, if the SLD-tree of $P \cup \{G\}$ contains a successful branch with a computed answer substitution θ then the pruned SLD-tree of $P \cup \{G\}$ contains a successful branch with a computed answer substitution θ' verifying $G\theta' \leq G\theta$.*

- *A loop check is* **shortening** *iff for every program P and goal G, if the SLD-tree of $P \cup \{G\}$ contains a successful branch with a computed answer substitution θ then the pruned SLD-tree of $P \cup \{G\}$ contains a successful branch, that is not longer, with a computed answer substitution θ' verifying $G\theta' \leq G\theta$.*

Another desired property of a loop check is that it prunes every infinite branch. This corresponds to the notion of completeness.

Definition 2.2 (Completeness)
A loop check is **complete** *iff it prunes every infinite SLD-derivation.*

Different loop checks will be presented in this chapter. One natural way of comparing them is to test whether if one detects a loop the other one detects this loop too.

Definition 2.3 (Stronger than)
The loop check L_1 is **stronger than** *the loop check L_2, iff for every program P and goal G, if the SLD-derivation $G \Rightarrow G_1 \Rightarrow \cdots$ is pruned by L_2 at the goal G_k then it is pruned by L_1 at a goal G_i with $i \leq k$.*

The following results are proved in [BAK 89]:

Proposition 2.1 *Let L_1 and L_2 be loop checks and L_1 be stronger than L_2.*

1. *If L_1 is weakly sound, then L_2 is weakly sound.*

2. *If L_1 is sound, then L_2 is sound.*

3. *If L_1 is shortening, then L_2 is shortening.*

4. *If L_2 is complete, then L_1 is complete.*

In the following sections, the concrete loop checks implemented will be presented.

2.2 Equality checks

A first class of loop check, named the *equality checks*, consists of understanding "G' is sufficiently similar to G" as "G' is equal to G" in one of the following two ways:

1. $G' = G\tau$ for some renaming τ (G' is a *variant* of G)

2. $G' = G\sigma$ for some substitution σ (G' is an *instance* of G)

The induced pruning is justified as follow. Assume that in the search tree there is a branch with a descendant node of G; say G', "sufficiently similar". Two cases must be considered:

- there exist a proof for G' so we can apply it directly to G (with some modification because G' is not exactly G). It is not necessary to prove G', we can prune G';

- there is no proof for G', so the attempt to prove G via G' cannot succeed, so we can prune G'.

Finally, the first two equality checks (called *equality checks for goals*) are the *Equals Variant of Goal* and the *Equals Instance of Goal.*

Definition 2.4 Equals Variant of Goal (EVG)
A derivation

$$G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \cdots$$

is pruned at the goal G_k by the **Equals Variant of Goal** *check if k is the smallest level such that there exists i, $0 \leq i < k$, and a* **renaming** *τ such that $G_k = G_i\tau$.*

Definition 2.5 Equals Instance of Goal (EIG)
A derivation

$$G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \cdots$$

is pruned at the goal G_k by the **Equals Variant of Goal** *check if k is the smallest level such that there exists i, $0 \leq i < k$, and there is a* **substitution** *σ such that $G_k = G_i\sigma$.*

The following proposition is proved in [BAK 89].

Proposition 2.2 *The Equals Variant of Goal and Equals Instance of Goals loop checks are weakly sound.*

Let G_i and G_k ($i \leq k$) be two goals of a derivation with $G_i\tau = G_k$. To be sound an equality check must only prune G_k if the SLD-tree pruned at G_k gives the same solution as the unpruned SLD-tree. This is the case if

$$G_0\theta_1 \cdots \theta_i\tau = G_k\theta_1 \cdots \theta_k \ .$$

The construction $G_0\theta_1 \cdots \theta_i \leftarrow G_i$ is called a *resultant* and $G_0\theta_1 \cdots \theta_i$ is the *resultant head.*

Here are the two new loop checks with this additional condition (called *equality checks for resultants*).

Definition 2.6 Equals Variant of Resultant (EVR)
A derivation

$$G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \cdots$$

is pruned at the goal G_k by the **Equals Variant of Resultant** *check if k is the smallest level such that there exists i: $0 \leq i < k$, and a* **renaming** τ *such that $G_k = G_i\tau$ and $G_0\theta_1 \cdots \theta_k = G_0\theta_1 \cdots \theta_i\tau$.*

Definition 2.7 Equals Instance of Resultant (EIR)
A derivation

$$G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \cdots$$

is pruned at the goal G_k by the **Equals Instance of Resultant** *check if k is the smallest level such that there exist i: $0 \leq i < k$, and a* **substitution** σ *such that $G_k = G_i\sigma$ and $G_0\theta_1 \cdots \theta_k = G_0\theta_1 \cdots \theta_i\sigma$.*

The following results are proved in [BAK 89].

Proposition 2.3 *The Equals Variant of Resultant and Equals Instance of Resultant loop checks are shortening.*

The equality checks are not complete for all *function-free* programs (programs without function symbols). As an example consider the program P = {a ← a,s.}. The SLD-tree of $P \cup$ {← a} is

```
← a
|
← a,s
|
← a,s,s
|
⋮
```

The equality checks do not prune it because the size of the goal increases, so there are never two "equal" goals. The problem is that there is an infinity of s atoms produced and they are never selected.

A class of programs where this never happens can be pointed out: the restricted programs.

Definition 2.8 Restricted program
Let P be a program. A clause $A_0 \leftarrow A_1, \ldots, A_n$ $(n \geq 0)$ is called **restricted w.r.t.** *P if there is no recursive (direct or indirect) call to A_0 in $A_1, \ldots, A_{n-1}$. The only atom that can be a recursive call is A_n.*

A program P is called **restricted** *if every clause in P is restricted w.r.t. P.*

The following results are proved in [BAK 89].

Proposition 2.4
The equality checks are complete for function-free restricted programs.

Proposition 2.5

1. *equals instant of goal is stronger than equals variant of goal;*
2. *equals variant of goal is stronger than equals variant of resultant;*
3. *equals instant of goal is stronger than equals instant of resultant;*
4. *equals instant of resultant is stronger than equals variant of resultant;*

Example 2.1 *To illustrate the difference between the equality checks for goals and for resultants, let us consider the following program:*
$P = \{C1 :$ a(Y) $\leftarrow$ b(Y).
$C2 :$ a(1) $\leftarrow$ a(1).
$C3 :$ b(2) $\leftarrow$ a(Y).
$C4 :$ b(3). $\}$.

The SLD-tree of $P \cup \{\leftarrow$ a(X)$\}$ is shown in the figure 2.1. In this tree, the nodes are not the goals but the resultants to show the difference between loop checks for goals and for resultants. The place where a loop check prunes is marked by a line label by the name of the loop check.

The equality checks for goals are weakly sound so they prune too early for finding the solution $\{X/2\}$.

2.3 Subsumption checks

In the subsumption checks the relation "G' is sufficiently similar to G" is understood as G is included in G', modulo a renaming or a substitution. Similar to the equality checks, we consider four subsumption checks: two for goals and two for resultants.

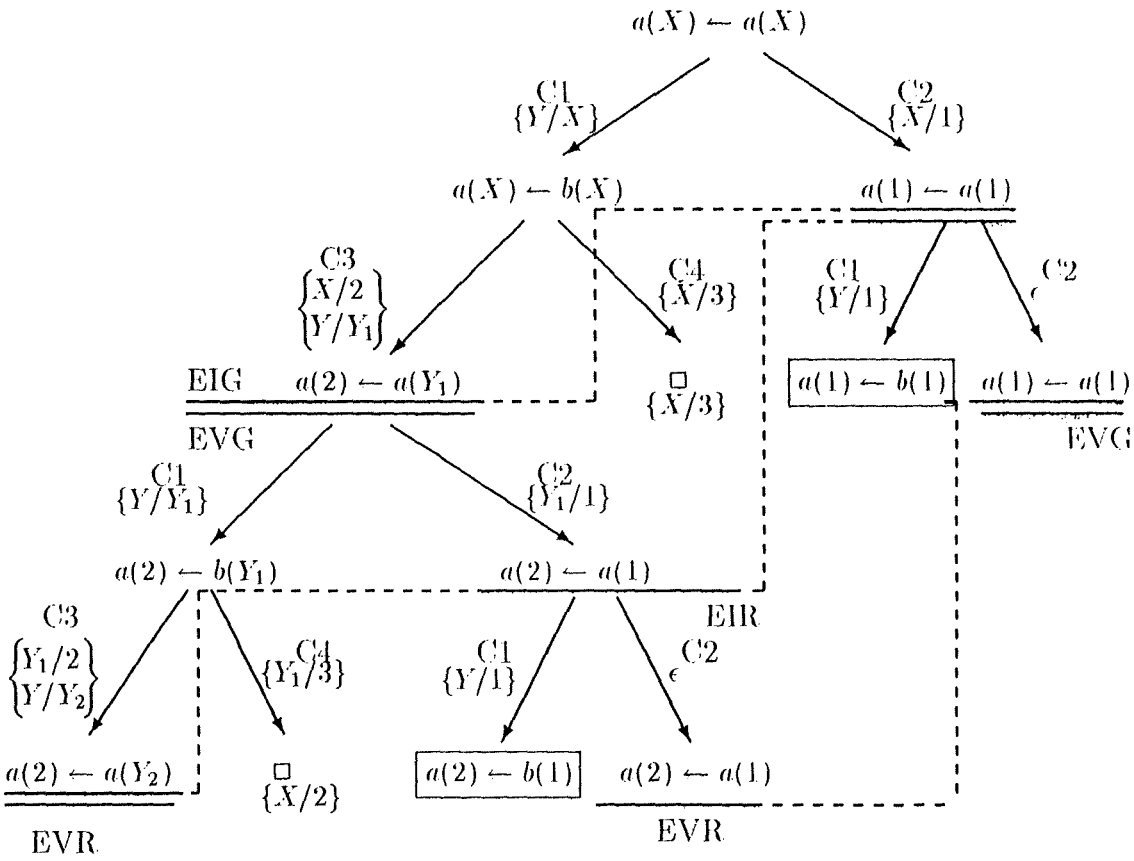

Figure 2.1: SLD-tree of $P \cup \{\leftarrow \mathsf{a(X)}\}$

Definition 2.9 Goal inclusion
For two goals G_1 and G_2, $G_1 \subseteq G_2$ if all the elements of the representation of G_1 as a list occur in the same order in the list representation of G_2, not necessary in adjacent positions.

For example $(a, c) \subseteq (a, b, c)$, but $(c, a) \not\subseteq (a, b, c)$. Here only the list representation of goal is used. In [BAK 89] the multiset representation of goal is also used, the distinction between lists and multisets plays also a role for the equality checks. If we use the multiset representation of goals then $(c, a) \subseteq (a, b, c)$.

Definition 2.10 Subsumes Variant of Goal (SVG)
A derivation

$$G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_i,\theta_k} G_k \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \cdots$$

is pruned at the goal G_k by the **Subsumes Variant of Goal** *check iff k is the smallest level such that there exists i: $0 \leq i < k$ and a* **renaming** *τ such that $G_k \supseteq G_i\tau$.*

Definition 2.11 Subsumes Instance of Goal (SIG)
A derivation

$$G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_i,\theta_k} G_k \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \cdots$$

is pruned at the goal G_k by the **Subsumes Instance of Goal** *check iff k is the smallest level such that there exists $i : 0 \leq i < k$ and a* **substitution** *σ such that $G_k \supseteq G_i\sigma$.*

Definition 2.12 Subsumes Variant of Resultant (SVR)
A derivation

$$G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_i,\theta_k} G_k \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \cdots$$

is pruned at the goal G_k by the **Subsumes Variant of Resultant** *check iff k is the smallest level such that there exists $i : 0 \leq i < k$ and a* **renaming** *τ such that $G_k \supseteq G_i\tau$ and $G_0\theta_1 \cdots \theta_k = G_0\theta_1 \cdots \theta_i\tau$.*

Definition 2.13 Subsumes Instance of Resultant (SIR)
A derivation

$$G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow_{C_i,\theta_k} G_k \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} G_k \Rightarrow \cdots$$

is pruned at the goal G_k by the **Subsumes Instance of Resultant** *check iff k is the smallest level such that there exist $i : 0 \leq i < k$ and a* **substitution** *σ such that $G_k \supseteq G_i\sigma$ and $G_0\theta_1 \cdots \theta_k = G_0\theta_1 \cdots \theta_i\sigma$.*

The following results are proved in [BAK 89].

Proposition 2.6

1. *subsumes variant of goal, subsumes instance of goal are weakly sound;*
2. *subsumes variant of resultant, subsumes instance of resultant are shortening.*

Proposition 2.7

1. *subsumes instant of goal is stronger than subsumes variant of goal;*
2. *subsumes variant of goal is stronger than subsumes variant of resultant;*
3. *subsumes instant of goal is stronger than subsumes instant of resultant;*
4. *subsumes instant of resultant is stronger than subsumes variant of resultant;*
5. *every subsumption checks is stronger than the corresponding equality check.*

$$\leftarrow \mathsf{a}(\mathsf{X}_0),\mathsf{b}(\mathsf{X}_0)$$
$$|$$
$$\leftarrow \mathsf{a}(\mathsf{X}_1),\mathsf{s}(\mathsf{X}_1,\mathsf{X}_0),\mathsf{b}(\mathsf{X}_0)$$
$$|$$
$$\leftarrow \mathsf{a}(\mathsf{X}_2),\mathsf{s}(\mathsf{X}_2,\mathsf{X}_1),\mathsf{s}(\mathsf{X}_1,\mathsf{X}_0),\mathsf{b}(\mathsf{X}_0)$$
$$|$$
$$\vdots$$
$$|$$
$$\leftarrow \mathsf{a}(\mathsf{X}_k),\mathsf{s}(\mathsf{X}_k,\mathsf{X}_{k-1}),\ldots,\mathsf{s}(\mathsf{X}_1,\mathsf{X}_0),\mathsf{b}(\mathsf{X}_0)$$
$$|$$
$$\vdots$$

Figure 2.2: SLD-tree of $P \cup \{\leftarrow \mathsf{a}(\mathsf{X}_0),\mathsf{b}(\mathsf{X}_0)\}$

Corollary 2.1 *All the subsumption checks are complete for function-free restricted programs.*

Subsumption checks are stronger than the corresponding equality checks, so it could be possible to find other classes of programs for which subsumption checks are complete. To suggest what can be these different classes let us analyse the following program $P = \{\mathsf{a}(\mathsf{X}) \leftarrow \mathsf{a}(\mathsf{Y}),\mathsf{s}(\mathsf{Y},\mathsf{X})\}$ with respect to the query $\leftarrow \mathsf{a}(\mathsf{X}_0),\mathsf{b}(\mathsf{X}_0)$. The SLD-tree of $P \cup \{\leftarrow \mathsf{a}(\mathsf{X}_0),\mathsf{b}(\mathsf{X}_0)\}$ is displayed in the figure 2.2.

It contains an infinite branch that is not pruned by any of the subsumption checks. As far as the subsumption checks for goals are concerned, one has to find a renaming or a substitution τ such that for one $i : 0 \leq i < k$

$$\mathsf{a}(\mathsf{X}_k),\mathsf{s}(\mathsf{X}_k,\mathsf{X}_{k-1}),\ \ldots\ ,\mathsf{s}(\mathsf{X}_1,\mathsf{X}_0),\mathsf{b}(\mathsf{X}_0)$$
$$\supseteq$$
$$(\mathsf{a}(\mathsf{X}_i),\mathsf{s}(\mathsf{X}_i,\mathsf{X}_{i-1}),\ \ldots\ ,\mathsf{s}(\mathsf{X}_1,\mathsf{X}_0),\mathsf{b}(\mathsf{X}_0))\,\tau$$

This amounts to finding a τ such that $\mathsf{X}_0\tau = \mathsf{X}_0, \ldots, \mathsf{X}_i\tau = \mathsf{X}_i$ and $\mathsf{X}_i\tau = \mathsf{X}_k$, which is impossible because there exists no τ such that $\mathsf{X}_i\tau = \mathsf{X}_i$ and $\mathsf{X}_i\tau = \mathsf{X}_k$. The resultant checks are weaker so we obtain the same result.

Why is this derivation not pruned? It's a combination of three problems:

1. a new variable is introduced by the "recursive" literal $\mathsf{a}(\mathsf{Y})$;
2. there is a relation between the new variable Y and the old variable X (via the literal $\mathsf{s}(\mathsf{Y},\mathsf{X})$);
3. the recursive literal $\mathsf{a}(\mathsf{Y})$ is selected before the other one $\mathsf{b}(\mathsf{X}_0)$.

The last problem is avoided by the use of restricted programs. We can define two new classes of programs that avoid the other two problems.

Definition 2.14 nvi program
A clause C is **non-variable introducing (nvi)** *iff every variable that appears in the body of C also appears in the head of C. A program P is* **nvi** *iff every clause in P is nvi.*

Definition 2.15 svo program
A clause C has the **single variable occurrence (svo)** *property iff in the body of C, no variable occurs more than once. A program P is* **svo** *iff every clause in P is svo.*

Clearly nvi programs avoid the first problem and svo programs avoid the second one. The following results are proved in [BAK 89].

Proposition 2.8 *Subsumption checks are complete for function-free nvi and svo programs.*

Example 2.2 *To illustrate where the different loop check prune, let us consider the following program P*

$P = \{C1 :$ a(X) $\leftarrow$ a(1), b(X).
$C2 :$ a(1).
$C3 :$ b(X) $\leftarrow$ c(X), b(Y).
$C4 :$ b(0).
$C5 :$ c(0). $\}$.

The SLD-tree of $P \cup \{\leftarrow$ a(Z)$\}$ is shown in figure 2.3. The equality checks do not prune the left branch because P is not a restricted program. This is only the case for the subsumption checks since they are complete for svo program and P is one of them.

The first loop check that prunes, among the loop checks we have studied, is the SIG check because a(1),b(Z) $\supseteq$ a(Z)τ *for the substitution $\tau = \{Z/1\}$. The next pruning loop check is the SVG check that prunes at two places: on one hand because* a(1),b(1),b(Z) $\supseteq$ (a(1),b(Z))τ *for the renaming $\tau = \epsilon$ and on the other hand because* c(Z),b(Y_2) $\supseteq$ b(Z)τ *for the renaming $\tau = \{Z/Y_2, Y_2/Z\}$. Finally the subsumption checks for resultants prune at two different places: the former one* a(1),b(1),b(Z) $\supseteq$ (a(1),b(Z))τ *and* a(Z) = a(Z)τ *for $\tau = \epsilon$ and the other one because* c(Y_2),b(Y_3) $\supseteq$ b(Y_2)τ *and* a(0) = a(0) τ *for $\tau = \{Y_2/Y_3\}$.*

2.4 Triangular loop checks

The equality and subsumption checks compare every goal with every previous goal. The number of goal comparisons performed is thus quadratic in the number of goals generated. For a finite SLD-derivation D , if $|D|$ is the length of the derivation D, then the equality and subsumption checks perform $\frac{1}{2}|D|(|D|-1)$ goal comparisons.

To improve the efficiency of loop checks, it is necessary to decrease the number of performed comparisons. In this section two loop checks derived from the previous loop checks are introduced. The understanding of the

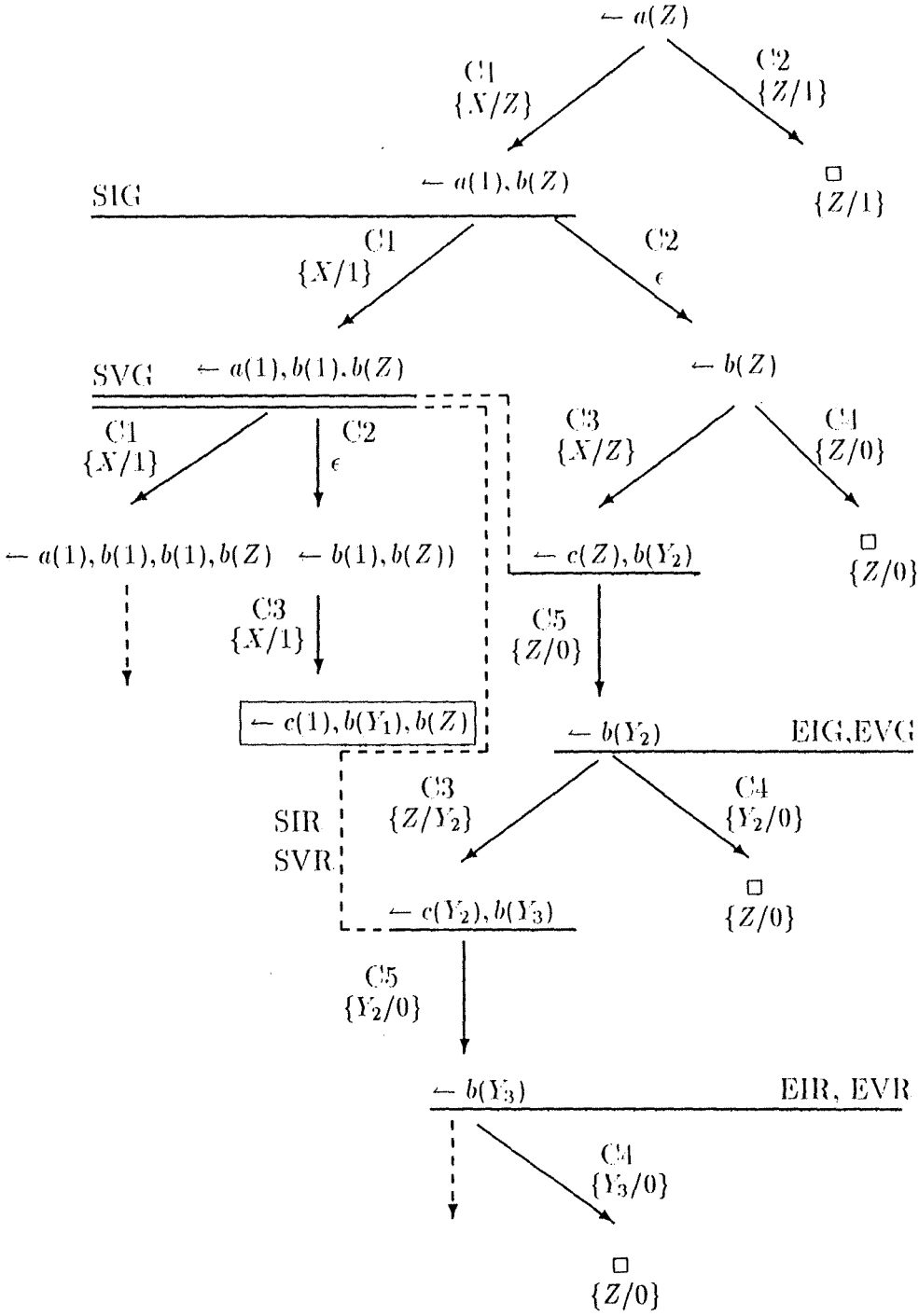

Figure 2.3: SLD-tree of $P \cup \{\leftarrow \mathsf{a(Z)}\}$

relation "is sufficiently similar" is not changed but the loop checks differ in where the comparisons are performed: given an infinite set of natural numbers S, two goals G_i and G_k are compared if $i < k$ and $k \in S$ (respectively $i, k \in S$).

Definition 2.16 single selected
A **single selected** *loop check S only compares two goals G_i, G_k $(0 \leq i < k)$ if $k \in S$.*

Definition 2.17 double selected
A **double selected** *loop check for S only compares two goals G_i, G_k $(0 \leq i < k)$ if $i, k \in S$.*

Note that by taking S as the set of natural number $\mathbb{N}$, the single and double selected loop checks become *full-comparison* loop checks (the one we had before).

Obviously, the number of comparisons depends on the set S. In [B90] it is shown for the double selected loop checks that if $S = \{\frac{1}{2}i(i+1) | i \in \mathbb{N}\}$ then the number of comparisons is linear in the number of goals developed.

Numbers of the form $\frac{1}{2}i(i+1)$ are usually called *triangular* numbers, this is why selected loop checks with $S = \{\frac{1}{2}i(i+1) | i \in \mathbb{N}\}$ are called *triangular loop checks.*

The following soundness and completeness results are also proved in [B 90].

Proposition 2.9 *For a fixed loop check:*

1. *full comparison is stronger than single selected;*
2. *single selected is stronger than double selected.*

So we have the following corollary.

Corollary 2.2

1. *The single and double selected equality and subsumption checks based on goals are weakly sound.*
2. *The single and double selected equality and subsumption checks based on resultants are shortening.*

Proposition 2.10

1. *The single and double selected equality checks are complete for function-free restricted programs.*
2. *The single and double selected subsumption checks are complete for function-free restricted programs.*
3. *The single and double selected subsumption checks are complete for function-free nvi programs.*
4. *The single and double selected subsumption checks are complete for function-free svo programs.*

Chapter 3

The Meta-interpreter

This chapter presents a meta-interpreter equiped with loop checking mechanisms for logic programs.

The loop checks are firstly implemented in a meta-interpreter because it is the easiest way to build a new PROLOG interpreter without having to write a huge program. The objective of this meta-interpreter is not to have an efficient implementation of the loop checks but rather to enable the practical comparison of the different loop checks.

This chapter is structured as follows. General meta-interpreters embodying no loop checking are presented in section 3.1 and some example of simple meta-interpreters are given in section 3.2. In section 3.3, the meta-interpreter that implement the loop checks presented in chapter 2 is described. Finally in section 3.4, the results obtained by the meta-interpreter are discused.

3.1 Introduction

Following [S 88], we define meta-interpreters as follows.

Definition 3.1 Meta-interpreter
A **meta-interpreter** *is an interpreter for (a subset of) a language written in the language itself.*

So meta-interpreters treat other program as data.

I'll only consider here meta-interpreters for PROLOG. In PROLOG the clauses of a program are represented as PROLOG terms and there exists several "non-first order predicates" to read or modify them. Therefore PROLOG does not make difference between program and data. Here this facility is used. The meta-interpreter is just added to the program and it read the program clauses when it needs them. Other solutions are possible, but then the program must be store in a file (or a database) and the access to program clauses must be explicitly programmed in the meta-interpreter, which is less efficient and needs more programming efforts.

Differences between meta-interpreters can be characterized in terms of their *granularity*, that is the parts of the computation that is made explicitly by the meta-interpreter.

The coarsest granularity is just to let PROLOG do everything: the goal to be solved is given to the interpreter. This meta-interpreter consist of only one clause:

```
solve(Goal) :-
        call(Goal).
```

where call(Goal) is a PROLOG meta-logical predicate that solve the goal Goal.

On the other hand the finest granularity is obtained by the meta-interpreter doing everything: it chooses the atom to be solved, the clause to be used, makes the unification, handles backtracking, ... The advantage of a finer granularity is that the meta-interpreter controls more (for example it can use an occur check in the unification) but of course such a meta-interpreter is slower, needs more memory space and requires much more programming effort.

In the remainder of this section different simple meta-interpreters are explained, these meta-interpreters can not handle full PROLOG: system predicates and ! are not treated correctly.

The "vanilla" meta-interpreter just selects the (leftmost) literal to be solved and let PROLOG do the remainder. Its code is given at figure 3.1.

```
solve(true).
solve((A,B)) :-
    solve(A), solve(B).
solve(A) :-
    clause(A,Body),
    solve(Body).
```

Figure 3.1: The "vanilla" meta-interpreter

To understand the "vanilla" interpreter, we have to explain the PROLOG predicate clause. clause(A, Body), where A is a non-variable term, search for a clause whose head matches A. The head and body of those clauses are unified with A and Body respectively. If the clause reduces to a fact then Body is the atom true, otherwise Body represents the body as a function with two argument ("first literal" and "remainder of the body") and the infixed functor ",".

Declaratively, the meta-interpreter acts as follows. The atom *true* is true (first clause). The conjunction (A,B) is true if A is true and B is true (second clause). A goal A is true if there is a clause A ← Body in the interpreted program such that Body is true (third clause).

Here is the procedural reading of the vanilla meta-interpreter. In the first clause, the empty goal, represented by the atom *true*, is solved. The next clause concerns goals with more than one literals, to solve a conjunction (A,B), solve A and then solve B. The goal reduction is covered by the final clause. To solve a goal, choose a clause from the program whose head unifies with the goal, and recursively solve the body of the clause.

```
solve([]).
solve([Atom|RestOfGoal]) :-
    clause(Atom, Body),
    append_body_to_goal(Body, RestOfGoal, NewGoal),
    solve(NewGoal).
append_body_to_goal(true, Goal, Goal) :-
    !.
append_body_to_goal((A,B), Goal, [A|B_Goal]) :-
    !,
    append_body_to_goal(B, Goal, B_Goal).
append_body_to_goal(B, Goal, [B]).
```

Figure 3.2: Meta-interpreter that keeps the entire goal to be solved as a list.

The procedural reading is necessary to show that the vanilla meta-interpreter reflects PROLOG's choices of implementing the abstract computation model of logic programming. The leftmost selection rule is guarantied by the second clause, the leftmost goal in the conjunction is solved first. The sequential search and backtracking comes from PROLOG's behaviour in satisfying the procedure clause.

This meta-interpreter does not keep explicitly the entire goal to be solved. So when we need it (we need it for the loop checking!), we have to maintain a list with the goal. The code of such a meta-interpreter is given in figure 3.2.

The central part of this meta-interpreter is the procedure
append_body_to_goal(Body, Goal, NewGoal)
where NewGoal is Body transformed into a list append to Goal:

if Body = *true*
    then
        NewGoal = Goal
    otherwise { Body = $(B_1, (B_2, (\ldots (B_{m-1}, B_m) \ldots)))$ $m \geq 1$}
        NewGoal = $[B_1, \ldots, B_m|$Goal$]$

The first clause is used if the clause was a fact (the body is represented by the atom *true*), then the list of body literals is the empty list, so NewGoal = Goal and it is the only solution. The cut (!) prevent to use the third clause that gives the wrong result: [*true*|Goal].

The second clause is used if the body has more than one literal, then the first one is the head of the list and the tail of the list is the remainder of the body, which is transformed in a list, is append to Goal, this is done by a recursive call to append_body_to_goal. The ! is also there to avoid the use of third clause, if the body has at least two literals. If the third clause is used with Body = (A,B) then the result is [(A,B)|Goal] and not [A,B|Goal].

The third clause is used if the body contains only one literal, then NewGoal is the list with this literal as first element followed by Goal.

Now for the solve procedure itself, the first clause is used if the goal to solve is empty, then the goal is solved. Otherwise we take the first element of the goal (because PROLOG uses the leftmost rules selection), and replace it by its body, which is done by append_body_to_goal. Again the mgu is applied implicitly. Finally the new goal, NewGoal, is solved. The sequential search and backtracking comes from PROLOG's behaviour in satisfying procedure clause.

3.2 Adding functionality to meta-interpreters

The meta-interpreters presented until now just execute logic programs and not more. But it is possible to add some side effects, for example tracing (printing each goal on the screen), producing the proof tree, stopping the research after D derivations, ...

Here is a meta-interpreter that stops the search if the proof depth is more than D [S 88] (which is in fact a meta-interpreter with a very simple loop check).

```
solve(true,D).
solve((A,B),D) :-
   solve(A,D),solve(B,D).
solve(A,D) :-
   D>0,
   clause(A,Body),
   D1 is D-1,
   solve(Body,D1).
```

It is based on the vanilla meta-interpreter. At each derivation step (in the third clause) the number of derivations left is decreased by one and the search stop when it reach 0.

Here is another interpreter that gives the number of derivation steps performed to find the solution.

```
solve(true,0).
solve((A,B),D) :-
    solve(A,DA),
    solve(B,DB),
    D is DA+DB.
solve(A,D) :-
    clause(A,Body),
    solve(Body,D1),
    D is D1 + 1.
```

The next interpreter can handle the system predicates ($is, <, >, =, \ldots$) [SS 86]. For that purpose we add a new clause that directly calls PROLOG to solve these predicates.

```
solve(true).
solve((A,B)) :-
    solve(A),solve(B).
solve(A) :-
    syst(A),
    call(A).
solve(A) :-
    clause(A,B),solve(B).
```

And we have to declare all the system predicates as

```
syst(clause(A,B)).
syst(X=Y).
syst(X is Y).
syst(X < Y).
      ⋮
```

Almost all meta-interpreters are based on the "vanilla"-interpreter or the meta-interpreter with goal represented as a list with some literals added, parameters and/or clauses to obtain the desired functionality.

3.3 The meta-interpreter with loop checks

In this section a meta-interpreter that performs loop checking is presented. First the information needed to perform loop checking and the choices made for their representation are explained, and then the meta-interpreter and the loop checking procedure are described. Finally, modifications are presented to make possible some prediction about more efficient interpreters with loop check.

3.3.1 The implementation choices

The implementation of loop checks described in chapter 2 requires at least the memorization of the list of previous goals and the current goal. Loop checks for resultants also need the list of the previous resultant heads and the current resultant head. Single and double selected loop checks need the current depth in the SLD-tree.

So our starting point is the meta-interpreter that explicitly represents the current goal as a list. It keeps two other lists, one with the previous goals and the other one with the previous resultant heads. In this meta-interpreter, the mgu found by the unification between the first literal of the goal and the head of the clause is not given explicitly, but found by the underlying PROLOG interpreter, and hence applied everywhere. But here this substitution must not be applied to the list of previous goals, but must only be applied to the current goal and used to compute the new resultant. Hence the unification must be made explicitly by the meta-interpreter and the mgu must be memorised. An additional advantage of making the unification in the meta-interpreter is that it allows us to opt for a sound unification procedure (with occur check).

Another problem in the previous meta-interpreters is that they do not make a difference between variables of the interpreted program (*object-variables*) and variables of the meta-interpreter (called *meta-variables* because they can be instantiated to a goal). To prevent this problem, the object-variables are replaced by new constants starting with the character "$" (a variable A become a constant $\$A$).

A binding is represented by a function with *eq* as functor and two arguments, the first one is the name of the variable and the second one is the term bound to that variable and a substitution is represented as a list of bindings, $\{x_1/t_1, \ldots, x_n/t_n\}$ becomes $[eq(x_1, t_1), \ldots, eq(x_n, t_n)]$.

3.3.2 The meta-interpreter

The user adds his program to the meta-interpreter and presents a goal of the form ←solve(Goal,Check) to it (the program must not use predicates defined by the meta-interpreter). The parameter Check specifies the kind of loop check that is to be used. The available values are *non* (no loop check and does not keep the lists of goals and resultant heads, only the usual interpretation), *empty* (no loop check but updates the lists of goals and resultant heads), for the full comparison *cvg*, *cig*, *cvr*, *cir*, *svg*, *sig*, *svr*, *sir* and for their single triangular *(***,st)* and double triangular *(***,dt)* variant. Goal is the goal to be solved (the root of the SLD-tree) where variables are represented by a string starting with "$".

The procedure solve1 initializes the counters used for the statistics, calls the meta-interpreter itself (with all the parameters) and prints the value of the counters and the derivation after it finds a solution. The counters are used to compare the number of unifications and matchings made to find the solution and those made by the loop check. The meta-interpreter stops and gives the value of the counters and the current branch of the SLD-tree when it finds a solution, when it prunes a branch and when it arrives at a goal that fails.

The main procedure of the meta-interpreter is

```
solve(Check, Goal, Resultant, Substitution, ListGoal, ListResult,
    LastVar, Depth, I, FI, Derivation).
```

It has eight input parameters and one output parameter. When the derivation

$$G_0 \Rightarrow_{C_1,\theta_1} \cdots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k$$

has been constructed, these parameters have the following meaning.

Input parameters for the construction of the derivation:

Goal	G_k (also used for loop checking);
Substitution	$\theta_1 \cdots \theta_k$, restricted to the variables of G_0;
LastVar	the number of variables used (needed for standardization apart).

Input parameters for loop checking:

```
solve(Check, [], _, _, _, _, _, _, _, _, [true]) :-                    /* success */
    {prints the computed answer substitution and the length of the derivation}.
solve(Check, Goal, Resultant, Substitution, ListGoal, ListResult,
        LastVar, Depth, I, FI, Derivation) :-
    check(Check, Goal, Resultant, ListGoal, ListResult, Depth, FI,
        NewListGoal, NewListResult),
    !,                                          /*no loop has been detected */
    find_new_goal_resultant(Goal, Resultant, Substitution, LastVar,
        NewGoal, NewResultant, NewSubstitution, NewLastVar),
    ( NewGoal = [false|_] - >
                                        /* the current goal has no resolvent */
        Derivation = [false],
        {prints the length of the derivation}
    ;                                   /* the current goal has a resolvent */
        update_depth(Depth, I, FI, NewDepth, NewI, NewFI),
        Derivation = [NewGoal|NewDerivation],
        solve(Check, NewGoal, NewResultant, NewSubstitution, NewListGoal,
            NewListResult, NewLastVar, NewDepth, NewI, NewFI, NewDerivation)
    ).
solve(Check, Goal, _, _, _, _, _, _, [prune]) :-                  /* a loop : prune */
    {prints the length of the derivation}.
```

Figure 3.3: The solve procedure

Check is the loop check that is used;

Resultant is $G_0\theta_1 \cdots \theta_k$;

ListGoal is $[G_{k-1}, \ldots, G_0]$[1];

ListResult is $[G_0\theta_1 \cdots \theta_{k-1}, \ldots, G_0]$[12];

Depth is k;

$FI = \frac{1}{2}I(I+1)$ and $\frac{1}{2}(I+1)I < Depth \leq \frac{1}{2}I(I+1)$.

Output (provided that a finite derivation is generated):

Derivation:	$[G_{k+1}, \ldots, G_n, true]$	means $G_n = \Box$,
	$[G_{k+1}, \ldots, G_n, prune]$	means G_n is pruned and
	$[G_{k+1}, \ldots, G_n, false]$	means G_n has failed.

The figure 3.3 presents a simplified version of the procedure solve without the output predicates which print some information on the screen. The complete version of the meta-interpreter can be found in the appendix.

[1]When a double triangular loop check is used, these lists contain only the goals (resultant heads) with a triangular index. When non is used, these lists are not maintained.

[2]The list of resultant heads is maintained only for loop checks for resultants and for the *empty* loop check.

The first clause is used when the current goal is empty, that is when the meta-interpreter has found a solution. In that case the complete version prints the associated computed answer substitution together with the length of its derivation.

The second clause invokes the loop check procedure itself. If check succeeds, no loop is detected, find_new_goal_result computes the next goal to be solved. If the current goal has no resolvent, there is no clause head that unifies with the selected (leftmost) atom, then a list of the form [false|_] is produced as the next goal. So if the next goal is of the form [false|_] then current goal failed and Derivation is [false] otherwise the current goal is solved by a recursive call to solve. This is done by the control sequence P− >Q;R which is analogous to "if P then Q" else R. If a loop is detected in the second clause, check fail, then the third clause applies. In the full version the depth of the pruned leaf is printed.

3.3.3 Loop checking

The procedure check, which performs the loop checking itself, is in fact a kind of selector that calls the right loop check procedure according to the Check parameter and updates the lists of the previous goals and resultants. This procedure has 11 clauses, one for each loop checking mechanism implemented.

The updating of the list of goals (and the list of resultant heads for the checks for resultants) used for the loop check is the same for all the loop checks, except for the double triangular loop check. For the double triangular loop check we only put the current goal and resultant in the lists if the current depth (D) is in the set $S = \{\frac{1}{2}i(i+1) | i \in \mathbb{N}\}$. To know whether D is in the set S, three counters are maintain, D, i and fi, with $fi = \frac{1}{2}i(i+1)$ and $\frac{1}{2}(i-1)i < D \leq \frac{1}{2}i(i+1)$. When $D = fi$, D is in the set S.

For the other loop checks, we just add the current goal (and current resultant for the checks for resultants) to the lists every time.

The loop checks use five procedures: compute_substitution, renaming, incl_sub, incl_ren, same_substitution.

compute_substitution(E,F,Substitution) checks whether E is an instance of F, $E\sigma = F$. If so, it returns the substitution σ,Substitution.

renaming(E,F,Renaming) checks whether F is a variant of E, F= Eτ. If so, it returns the substitution τ (Renaming).

Note that, this is not a renaming but the renaming restricted to the variables of E. For example if E = a(X,Y,A) and F = a(A,B,C) then $\tau = \{X/A, Y/B, A/C\}$ and not $\{X/A, Y/B, A/C, C/X, B/Y\}$. This substitution is enough, the only time we need τ is in the variant checks for resultants, when we compare $G_0\theta_1 \cdots \theta_i \leftarrow G_i$ and $G_0\theta_1 \cdots \theta_k \leftarrow G_k$ $(i < k)$. We want to know if there exists a renaming τ, such that $G_i\tau = G_k$ and $G_0\theta_1 \cdots \theta_i\tau = G_0\theta_1 \cdots \theta_k$. We test this in three steps.

1. $\tau_1 = \{x_1/t_1, \ldots, x_n/t_n\}$ such that $G_i\tau_1 = G_k$;

2. $\tau_2 = \{y_1/s_1, \ldots, y_m/s_m\}$ such that $G_0\theta_1 \cdots \theta_i\tau_2 = G_0\theta_1 \cdots \theta_k$;

3. if for $j \in \{1, \ldots, n\}$ and $l \in \{1, \ldots, m\}$ we have $x_j = y_l$ then $t_j = s_l$ must hold.

The only case when the three conditions are satisfied and the renaming τ does not exist is when for example we have

$$G_0\theta_1 \cdots \theta_i \leftarrow G_i \equiv \mathsf{g(C)} \leftarrow \mathsf{f(B)}$$

and

$$G_0\theta_1 \cdots \theta_k \leftarrow G_k \equiv \mathsf{g(A)} \leftarrow \mathsf{f(A)}$$

where $\tau_1 = \{B/A\}$ and $\tau_2 = \{C/A\}$ satisfy the third condition. But this situation never occurs because the resultant head g(C) transforms into g(A) only if C/A $\in \theta_{i+1} \cdots \theta_k$. C does not appear in the variables of the clauses used to find the derivation from G_i to G_k by standardizing apart. So C must be in G_i and hence in the domain of τ_1.

The next two procedures are used for the subsumption checks. incl_sub(E,F,-Substitution) checks if there exists a substitution, Substitution, such that applied to E, E is included in F.

incl_ren(E,F,Renaming) checks whether there exists a renaming, Renaming, such that applied to E, E is included in F.

same_substitution(S1,S2) where S1 and S2 are two substitutions: S1 = $[eq(V_1, A_1), \ldots, eq(V_n, A_n)]$, S2 = $[eq(W_1, B_1), \ldots, eq(W_m, B_m)]$. It checks if for all $1 \leq i \leq n$ and $1 \leq j \leq m$ such that $V_i = W_j$ then $A_i = B_j$.

3.3.4 A more realistic meta-interpreter

The question "How costly is loop checking?" suggest a comparison between an efficient PROLOG interpreter with loop checking and existing PROLOG interpreters. As developing a really efficient interpreter with loop checking involves much more work than making the relatively simple implementation presented in this thesis, it would be helpful to have a meta-interpreter where the efficiency (or inefficiency) of the loop checking and the goal derivation are the same.

It must be possible to make a more efficient loop checking procedure if we are not forced to memorize twice the list of goals: one for the loop checking and the other one for the derivation mechanism. It must be also possible to find faster if there is a goal sufficiently similar to the current one without comparing the current goal with every previous goals. But to do this we have to write a completely different meta-interpreter with other data structure.

The meta-interpreter presented until now have the advantage to show all the SLD-tree even the pruned and failed branch which is not done by a "real" interpreter and is quite consuming in time and space.

To have a meta-interpreter that stop only when it finds a solution we have to change slightly the specification of solve and find_new_goal_result.

For

```
solve(Check, [], _, _, _, _, _, _, _, _) :-                    /*success*/
    {prints the computed answer substitution}
solve(Check, Goal, Resultant, Substitution, ListGoal, ListResult,
        LastVar, Depth, I, FI) :-
    check(Check, Goal, Resultant, ListGoal, ListResult, Depth, FI,
        NewListGoal, NewListResult),
    !,                                      /* no loop has been detected */
    find_new_goal_result(Goal, Resultant, Substitution, LastVar, NewGoal,
        NewResultant, NewSubstitution, NewLastVar),
    update_depth(Depth, I, FI, NewDepth, NewI, NewFI),
    solve(Check, NewGoal, NewResultant, NewSubstitution, NewListGoal,
        NewListResult, NewLastVar, NewDepth, NewI, NewFI).
```

Figure 3.4:

```
solve(Check, Goal, Resultant, Substitution, ListGoal, ListResult,
    LastVar, Depth, I, FI).
```

the meaning of the eight input parameters are the same as before. In this new version the output parameter has disappeared and it succeed only when a solution is found. Its code is given in the figure 3.4

The first clause is used, like in the previous version, when the current goal is empty.

The second clause invokes the loop check procedure itself. If check succeeds, no loop is detected, find_new_goal_result computes the next goal to be solved. If the current goal has no resolvent then find_new_goal_result fails (this is the difference with the first version of find_new_goal_result). If the current goal has a resolvent then it is solved by a recursive call to solve.

It is also possible to increase the efficiency of the goal derivation by using PROLOG variable in the program. Then the application of the substitution is done implicitly to all the terms. In order to avoid the substitution to be applied to the list of previous goals and resultant heads, goal (resultant head) that is added to the list of previous goals (previous resultant heads) is renamed first, using fresh variables.

I do not use this technique here because I think this is too efficient if we compare it with the loop checking where the matching is made completely explicitly. For example if we use the transitive closure program presented in the first chapter with the relation {r(a,b),. . .,r(y,z)}. and the meta-interpreter that use our variables, the "$ variables", the cpu-time to compute the derivations (the time spent in the procedure find_new_goal_resultant) is six time the cpu-time use to compute the same derivation with the meta-interpreter that use directly the PROLOG variables, see table 5.1. But the time used to make the loop checking itself (the cpu-time spent in the check procedure) is almost the same in the two meta-interpreter.

This will be discus in more detail in chapter 4.

3.4 The results

In this section, I describe the counters and some examples to show the results given by the first version of the meta-interpreter.

The first two counters report, respectively, the cpu-time used to construct the derivation (the time spent in the procedure find_new_goal_result), the cpu-time used to make the loop checking (the time spent in the procedure check). This is the cpu-time given by PROLOG. Note that its value changes from one execution to another, depending of the status of the system (the other processes) when the program is executed.

The other counters are the number of nodes of SLD-tree already developed, the number of goal comparisons performed in the loop check and the number of goals added to the list of previous goals and the list of previous resultant heads.

In PROLOG there are no global variables and if the results are passed by parameters, the information is lost when backtracking occurs. However, I use the Quintus PROLOG which offers the possibility to call some procedures written in another language. Thus the counters are implemented with global variables in C. There is 10 counters, five (0 ...4) that use integer numbers and five (0 ...4) that use real numbers. There is six predicates to manipulate them:

init_int(Counter): the integer counter Counter is equal to 0;

init_float(Counter): the real counter Counter is equal to 0;

add_int(Counter,Value): the integer Value is added to the integer counter Counter;

add_float(Counter,Value): the real Value is added to the real counter Counter;

read_int(Counter,Value): the value of the integer counter Counter is put in Value;

read_float(Counter,Value): the value of the real counter Counter is put in Value.

Let us take some examples of the second chapter to see what the output of the meta-interpreter looks like and how it can be translated back to an SLD-tree.

With program of example 2.1 (using equality checks), the following answers are produced.

```
?- solvel(eig,[a($X)]).
      derivation [[a($X)],[b($X)],[a($Y2)],prune]

      computed answer substitution [eq($X,3)]
      derivation [[a($X)],[b($X)],[],true]

      derivation [[a($X)],[a(1)],prune]
```

```
?- solvel(evr,[a($X)]).
        derivation [[a($X)],[b($X)],[a($Y2)],[b($Y2)],[a($Y4)],prune]

        computed answer substitution [eq($X,2)]
        derivation [[a($X)],[b($X)],[a($Y2)],[b($Y2)],[],true]

        derivation [[a($X)],[b($X)],[a($Y2)],[a(1)],[b(1)],false]

        derivation [[a($X)],[b($X)],[a($Y2)],[a(1)],[a(1)],prune]

        computed answer substitution [eq($X,3)]
        derivation [[a($X)],[b($X)],[],true]

        derivation [[a($X)],[a(1)],[b(1)],false]

        derivation [[a($X)],[a(1)],[a(1)],prune]
```

As shown in figure 2.1 the eig loop check finds only one solution, $\{X/3\}$ and prune the tree two times. The evr loop check finds two solutions, $\{X/2\}$ and $\{X/3\}$; prunes three times and find a leaf that failed.

And for the program of example 2.2 (using subsume checks), the following answers are produced.

```
?- solvel(sig,[a($Z)]).
        Derivation = [[a($Z)],[a(1),b($Z)],prune];

        computed answer substitution: [eq($Z,1)]
        Derivation = [[a($Z)],[],true]

?- solvel(svg,[a($Z)]).
        Derivation = [[a($Z)],[a(1),b($Z)],[a(1),b(1),b($Z)],prune];

        Derivation = [[a($Z)],[a(1),b($Z)],[b($Z)],[c($Z),b($Y2)],prune];

        computed answer substitution: [eq($Z,0)]
        Derivation = [[a($Z)],[a(1),b($Z)],[b($Z)],[],true];

        computed answer substitution: [eq($Z,1)]
        Derivation = [[a($Z)],[],true]

?- solvel(sir,[a($Z)]).
        Derivation = [[a($Z)],[a(1),b($Z)],[a(1),b(1),b($Z)],prune];

        Derivation = [[a($Z)],[a(1),b($Z)],[b($Z)],[c($Z),b($Y2)],[b($Y2)],
                [c($Y2),b($Y4)], prune];

        computed answer substitution: [eq($Z,0)]
```

```
Derivation = [[a($Z)],[a(1),b($Z)],[b($Z)],[c($Z),b($Y2)],[b($Y2)],[],true];

computed answer substitution: [eq($Z,0)]
Derivation = [[a($Z)],[a(1),b($Z)],[b($Z)],[],true];

computed answer substitution: [eq($Z,1)]
Derivation = [[a($Z)],[],true]
```

The result of this program for the equality checks is not displayed because they do not prune the first branch which is infinite, so they never give a solution. As shown in figure 2.3 the sig loop check finds only one solution, $\{Z/1\}$ and prunes once. The svg loop check finds two solutions, $\{Z/0\}$ and $\{Z/1\}$ and prunes two times. The sir loop check finds three solutions, $\{Z/0\}$ two times and $\{Z/1\}$ and prunes two times.

Chapter 4

The pre-compiler

This chapter presents a pre-compiler that transforms a PROLOG program into another one with loop checking mechanisms included in it.

The loop checks are implemented almost in the same way as in the meta-interpreter, but the transformed program executed under PROLOG is more efficient than the program interpreted by the meta-interpreter. The main reasons are that the additional layer of interpretation is removed and the search for applicable clause and unification is then done by the underlying PROLOG system, but the computations performed for the loop checks are still explicit.

4.1 Presentation of the pre-compiler

The purpose of the pre-compiler is to take a program (here a PROLOG program) and to transform it into another program of the same language by adding instructions to improve or modify the execution of the original program (here a PROLOG program that uses a loop checking mechanism). Then the resulting program can be interpreted or compiled like every program (here it is interpreted by the ordinary PROLOG interpreter). This is schematized in figure 4.1.

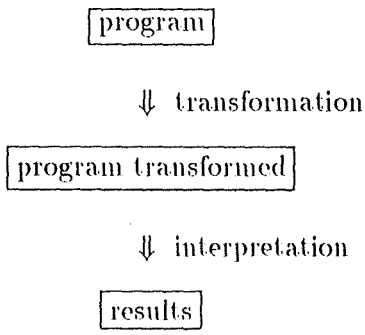

Figure 4.1:

```
append([ ],L2,L2).                          append([ ], L2, L2,1).
append([H|T],L2,[H|LApp]) :-      ⇒       append([H|T], L2, [H|LApp], LA) :-
    append(T,L2,LApp).                          append(T,L2,LApp,LA1),
                                                LA1 is LA + 1.
```

Figure 4.2: transformation of the append procedure to compute the length of the derivation.

Example 4.1
An example of a pre-compiler transformation consist in adding new parameters and literals to every clauses of the program in order to compute the length of the derivation. Two parameters need to be added to every literal for that purpose: the first one is the length of the derivation before the literal is proved and the second is the length of the derivation after the success of the literal. Furthermore, an extra instruction is to be added at the beginning of each clause: it consists in adding 1 to the length of the derivation. Figure 4.2 gives an example of this transformation to the append *procedure.*

After the resolution of append([1, 2, 3], [4, 5], LApp, Length), Length *is the length of the derivation to find* Lapp = [1, 2, 3, 4, 5], *the concatenation of the lists* [1, 2, 3] *and* [4, 5].

It is possible to execute the transformed program with the usual PROLOG interpreter and then to compare its execution with the program before its transformation. The transformation can be done automatically and once for all.

4.2 The pre-compiler for loop checking

The transformation of a program without loop check to one with loop check consists of adding a call to the procedure loop_check in each clause and adding the parameters needed for the loop checking in each literal. To execute the transformed program it has to be loaded in memory with the loop_check procedure. Then it can be directly executed with the PROLOG interpreter. Because the goal derivation is done by the underlying PROLOG system it is not possible to stop when a derivation fails like with the meta-interpreter. Thus the transformed program only stop when it finds a solution and not when it prunes or finds a failed derivations.

4.2.1 Representation of the objects

To be able to use the PROLOG interpreter the normal PROLOG variables are used, and not our own variables as in the meta-interpreter. We must still avoid that a variable that is bound by solving the current goal is also bound in the list of previous goals (resultant heads). To this end each goal (resultant head) that is added to the list of previous goals (previous resultant heads) is renamed first, using fresh variables (of course the same renaming must be applied to a goal G_k and the corresponding resultant head $G_0\theta_1 \cdots \theta_k$). This renaming is done by the procedure copy_term.

4.2.2 The transformation

The parameters to be added to the literals are of three kinds:

the loop check to be performed;

the parameter values needed by the loop check before the execution of the clause;

the parameter values after the execution of the clause.

As seen in subsection 3.3.1, the pieces of information needed by the loop check are the following: the kind of loop check to be performed, the current goal, the list of the previous goals, the current resultant head, the list of the previous resultant heads and the current depth in the derivation (for the selected loop checks).

If we have the following derivation

$$G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{j-1} \Rightarrow_{C_j,\theta_j} G_j = \mathsf{h}(\bar{X}),\mathsf{l}_1(\bar{Z}_1),\ldots,\mathsf{l}_n(\bar{Z}_n) \Rightarrow \cdots$$

$$\Rightarrow G_{m-1} \Rightarrow_{C_m,\theta_m} G_m = (\mathsf{l}_1(\bar{Z}_1),\ldots,\mathsf{l}_n(\bar{Z}_n))\theta_{j+1}\cdots\theta_m \Rightarrow \cdots$$

such that no loop is detected and no goal without resolvent are found in the sequence $G_j \Rightarrow \cdots \Rightarrow_{c_{m-1},\theta_{m-1}} G_{m-1}$ then the call to the procedure h in the transformed program to solve $\mathsf{h}(\bar{X})$ in G_j with the loop check Check is

h($\bar{X}$, Check, ListGoal, ListResult, Goal, Resultant, Depth, ListGoal',
ListResult', Goal', Depth')

when the clause is entered

ListGoal is $[G_j,\ldots,G_0]$[1];

ListResult is $[G_0\theta_1\ldots\theta_j,\ldots,G_0]$[12];

Goal is G_j =[h($\bar{X}$),$\mathsf{l}_1(\bar{Z}_1),\ldots,\mathsf{l}_n(\bar{Z}_n)$];

Resultant is $G_0\theta_1\cdots\theta_j$;

Depth is j

and when the clause is done

ListGoal' is $[G_m,\ldots,G_0]$[1];

ListResult' is $[G_0\theta_1\ldots\theta_m,\ldots,G_0]$[12];

Goal' is G_m =([$\mathsf{l}_1(\bar{Z}_1),\ldots,\mathsf{l}_n(\bar{Z}_n)$]) $\theta_{j+1}\cdots\theta_m$;

Resultant is $G_0\theta_1\cdots\theta_j\theta_{j+1}\cdots\theta_m$;

[1] When a double triangular loop check is used, these lists contain only the goals (resultant heads) with a triangular index. When non is used, these lists are not maintained.

[2] The list of resultant heads is maintained only for loop checks for resultants and for the *empty* loop check

Depth' is m.

The resultant head (Resultant) appears only once in the parameters because it is always the same term except that the substitution $\theta_{j+1} \ldots \theta_m$ is applied. PROLOG interpreter applies these substitutions to every parameters. Therefore, if before the clause is entered Resultant is $G_0\theta_1 \cdots \theta_j$ then when the clause is done Resultant is $G_0\theta_1 \cdots \theta_j\theta_{j+1} \cdots \theta_m$.

If a loop is detected or there is a goal without resolvent in the sequence $G_j \Rightarrow \cdots \Rightarrow_{c_{m-1},\theta_{m-1}} G_{m-1}$ then the call to the procedure h fails.

loop_check is the only literal added in the body of the clause to perform the loop checking. It is added as the first literal of the body. If we have the following clause:

h($\bar{X}$) :-
    $b_1(\bar{Y}_1)$,
    ⋮
    $b_n(\bar{Y}_n)$.

then the transformed clause will be:

h($\bar{X}$, Check, ListGoal, ListResult, Goal, Resultant, Depth,
        ListGoal_{n+1},ListResult_{n+1},Goal_{n+1},Depth_{n+1}) :-
    loop_check(Check, [$b_1(\bar{Y}_1)$, ..., $b_n(\bar{Y}_n)$], ListGoal, ListResult, Goal, Resultant,
        Depth, ListGoal_1, ListResult_1, Goal_1,Depth_1),
    $b_1(\bar{Y}_1$, Check, ListGoal_1, ListResult_1, Goal_1, Resultant,
        Depth_1, ListGoal_2, ListResult_2, Goal_2, Depth_2),
    ⋮
    $b_n(\bar{Y}_n$, Check, ListGoal_n, ListResult_n, Goal_n, Resultant, Depth_n,
        ListGoal_{n+1}, ListResult_{n+1}, Goal_{n+1}, Depth_{n+1}).

4.2.3 The loop_check procedure

loop_check updates the current goal, the list of goals, the list of resultant heads and the depth; checks if it detects a loop or not. If it detects a loop it prints that it prunes the tree, the current list of goals and depth and then fails. If it has found a solution it prints the current list of goals and depth. loop_check has seven input parameters:

Check: the kind of loop check to be performed;

[$b_1(\bar{Y}_1)$, ... ,$b_n(\bar{Y}_n)$]: the body of the clause;

ListGoal: the list of previous goals without the current one;

ListResult: the list of previous resultant heads without the current one;

Goal: the previous goal, the goal with h($\bar{X}$) as first literal;

Resultant: the current resultant head;

```
loop_check(Check, [], ListGoal, ListResult, [H], Resultant, Depth,
        [[H]|ListGoal], [Resultant|ListResult], [H], NewDepth) :-
    !,
    copy_term([Resultant|[H]], [NewResultant|NewGoal]),
    NewDepth is Depth + 1,
    {prints that it found a solution, the derivation and the length of the derivation}.
loop_check(Check, Body, ListGoal, ListResult, [H|Goal], Resultant, Depth,
        NewListGoal, NewListResult, CurrentGoal, NewDepth) :-
    NewDepth is Depth + 1,
    append(Body, Goal, CurrentGoal),
    check(Check, CurrentGoal, Resultant, ListGoal, ListResult, NewDepth,
        NewListGoal, NewListResutl),
    !.                                          /* no loop has been detected*/
loop_check(Check, Body, ListGoal, ListResult, [H|Goal], Resultant, Depth,
        _, _, CurrentGoal, NewDepth) :-
    {prints that it prunes the SLD-tree, prints the derivation and its length}
    !,                                                  /* there is a loop */
    fail.
```

Figure 4.3: The loop_check procedure

Depth: the depth of Goal in the derivation

and four output parameters

$ListGoal_1$: the list of previous goal with the current one;

$ListResult_1$: the list of previous resultant heads with the current one;

$Goal_1$: the current goal;

$Depth_1$: the current depth in the derivation

A simplified version of the loop_check procedure is given in figure 4.3. This is a version without the predicates that display the results on the screen. The full version can be found in the appendix. The procedure loop_check has three clauses.

The first clause applies when a solution is found, the current goal is empty (the body of the clause is empty and the previous goal contain only the head of the clause). It does not look for a loop.

If the current goal is not empty the second clause is invoked. It computes the current goal and depth and calls the check procedure (to check for a loop). The check procedure is almost the same as the one in the meta-interpreter (its text is in the appendix). If check does not detect a loop, then it succeeds.

If check failed, a loop is detected, the third clause is invoked. The third clause prints that it prunes the tree at this point and prints the list of goals and the depth. If the user is not interested in knowing when the tree is pruned, this clause can be removed.

4.2.4 The pre_compiler program

The transformation of the program, to add the loop checking mechanism, is quite easy but it's tedious because of the number of parameters to add to all the clauses. Here is a program (a "pre-compiler") which make this transformation automatically.

Another problem is that with the transformed program, when the user wants to ask a query, he has to initialize the parameters for the loop check. The solution is to provide a new clause that fills all these new parameters for him. This new clause has the same name as the old one and the same parameters plus a new one which is the kind of loop checking to be performed. If $h(\bar{X})$ is a predicate the user will use in the query, a new clause is added:

```
h(X, Check) :-
    copy_term([h(X)], OldHeadRename),
    h(X, Check, [OldHeadRename], OldHeadRename, [h(X)], h(X), 0,
        ListGoal, ListResult, LastGoal, Depth).
```

This works only if the user asks only queries that consist of one atom!

The arguments of the pre_compile procedure are:

the file where the program to be transformed is;

the file where it puts the transformed program;

the name of the procedure that the user can use as a query;

the arity of this procedure (without the loop checking argument).

The procedure pre_compile opens the input and output file, creates the clause that the user can use as query (the one with the same parameters as the original one plus the parameter for the kind of loop check) and transforms the program. To make the transformed program it calls the transform_clause procedure which transforms all the clauses from the current clause to the end of the input file and puts them into the output file.

The first clause of transform_clause procedure is used when it is at the end of the input file (the current clause is "end_of_file").

The second one is used if the current clause has a body, it is a function with two argument (the head and the body) and ":-" as functor. It makes the head of the new clause by taking the old one and adding the extra arguments. To make the body of the new clause, it puts the call to the loop_check procedure as the first literal of the body and then the literals of the original body, where it adds the extra parameters. It reads the next clause and calls recursively transform_clause for transforming the remainder of the input stream.

The third one is used if the current clause has no body, it isn't a function with ":-" as functor (the test in the previous clause fail). The head of the

new clause is the head of the original one where we add the extra parameters. The body of the new clause has only one literal, the call to the procedure loop_check. It reads the next clause and call recursively transform_clause for transforming the remainder of the input stream.

4.3 Modification of the pre-compiler

The program transformations presented in the previous section, are quite simple, as they always do the same thing to all the clauses of the program.

But it is convenient to make other transformations. In this section three of them are presented: adding a unification with occur check, omitting the loop check in certain clauses (for which the user knows there is no loop) and allowing to use "built-in predicates" that cannot be transformed.

4.3.1 Sound unification

In the previous section, only the unification of PROLOG is used, without occur check. To add the occur check to the unification, it is possible to write a new predicate that performs the unification with occur check and use it when the occur check is necessary. Let us look where unification is used in a PROLOG program to see how to incorporate the sound unification. First, unification is used when the user asks it explicitly (i.e. X=Y). In this case it is up to him to decide if he needs the unification with or without occur check and we have to provide him with both of them. Thereafter, we will use X = Y for a unification without occur-check and unif(X, Y) for a unification with occur-check. The second case is the unification of a clause head with a literal. All the variables of the clause head are different from the variables of the literal. The occur check is unnecessary if the clause head has no repeated variables (each variable of the clause head unifies with one term that does not contain the variable itself). Heads without repeated variables are called *linear* [S 89]. We can transform nonlinear clause heads by replacing repeated occurrences of variables by new variables to make the clause head linear and the new variables in the transformed clause head are then unified with the original variables by sound unification with occur check in the transformed clause body. Now unifying the clause head with a literal can be done without occur check. The user desiring a unification with occur check can use the new predicate unif(X,Y).

This transformation can be done automatically by the pre-compiler. The principle is to find the repeated occurrences of variables in the head to replace them by new variable except one occurrence of each and then to add in the body calls to the predicate unif (which performs sound unification) to unify the new variables with the original one.

Example 4.2

```
                        p(X1, Y1, f(X,Y)) :-
p(X,Y,f(X,Y)).    ⇒         unif(X,X1),
                            unif(Y,Y1).
```

Of course, the pre-compiler will perform the other body transformation to include the loop check mechanism in the transformed program. The complete program transformation is decomposed into three parts. In the first one, the clause head is made linear, as just described. This is achieved by the procedure find_dupl.

The second one transforms the clause to include the loop checking mechanism.

And the third one, realized when we have the transformed body, is to add at the beginning of the body one call to unif per pair in the list of pairs (original variable, new variables). This is done by the procedure add_unif.

4.3.2 Omitting loop check in certain clauses

Another modification of the transformed program is aimed at decreasing the cost of loop checking. Sometimes we know that some procedure never generates an infinite loop. For these procedures it is a good idea not to perform the loop check, but to update the different arguments only. The easiest way to do that, is to use in these procedures the loop check *empty* to be able to continue the loop checking at the end of the procedure.

The user has to put the atom no_loop_check as the first literal of the body when he does not want to perform the loop check in that clause and the one that are called by it (the pre-compiler puts the loop check *empty* in that clause).

In the pre-compiler there are two different ways of transforming the body. If the first literal is the atom no_loop_check then the transformed body has the call to loop_check as first literal and then the literals different form no_loop_check transformed in the usual way, except that the kind of loop check is *empty*. Otherwise we transform the body in the usual way.

4.3.3 Built-in predicates

The original version of the pre-compiler transforms all the literals in the body, but if some "built-in predicate" are used as write, we cannot add the extra arguments to this predicate. The solution is to test before adding the extra arguments if the predicate is not a system predicate. This is done by the procedure transform_literal: if the predicate is declared as system (by the predicate system) then the new literal is the same, otherwise extra parameters are added to the original one.

4.4 Example

The result of transforming the program of figure 2.1:

```
a(Y) :- b(Y).
a(1) :- a(1).
b(2) :- a(Y).
b(3).
```

is:

```
/* new program created by transforming the file fig2.2 */
a(_20,_8):-
    copy_term([a(_20)],_11),
    a(_20,_8,[_11],_11,[a(_20)],a(_20),0,_12,_13,_14,_15).
a(_92,_117,_118,_119,_120,_121,_122,_123,_124,_125,_126):-
    loop_check(_117,[b(_92)],_118,_119,_120,_121,_122,_167,_168,_169,_170),
    b(_92,_117,_167,_168,_169,_121,_170,_123,_124,_125,_126).
a(1,_254,_255,_256,_257,_258,_259,_260,_261,_262,_263):-
    loop_check(_254,[a(1)],_255,_256,_257,_258,_259,_304,_305,_306,_307),
    a(1,_254,_304,_305,_306,_258,_307,_260,_261,_262,_263).
b(2,_392,_393,_394,_395,_396,_397,_398,_399,_400,_401):-
    loop_check(_392,[a(_374)],_393,_394,_395,_396,_397,_442,_443,_444,_445),
    a(_374,_392,_442,_443,_444,_396,_445,_398,_399,_400,_401).
b(3,_512,_513,_514,_515,_516,_517,_518,_519,_520,_521):-
    loop_check(_512,[],_513,_514,_515,_516,_517,_518,_519,_520,_521).
```

The transformed program is difficult to read because it uses the PROLOG notation for the variables (numbers preceded by a "_") and not the variables of the original program (strings with a capital letters as first character). This is due to the fact the pre-compiler takes the clauses of the program as PROLOG objects (variables, predicates, ...) and not as a string of characters. Here are the results of executing the transformed program.

```
?- a(X,eig).
        prune
        Derivation = [[a(_113)],[b(_71)],[a(_15)]]

        true
        Derivation = [[],[b(_71)],[a(_15)]]
        X = 3 ;

        prune
        Derivation = [[a(1)],[a(_15)]]
        no
?- a(X,eir).
        prune
        Derivation = [[a(_295)],[b(_250)],[a(_166)],[b(_71)],[a(_15)]]

        true
        Derivation = [[],[b(_250)],[a(_166)],[b(_71)],[a(_15)]]
        X = 2 ;

        prune
        Derivation = [[a(1)],[a(1)],[a(_166)],[b(_71)],[a(_15)]]
```

```
true
Derivation = [[],[b(_71)],[a(_15)]]
X = 3 ;

prune
Derivation = [[a(1)],[a(1)],[a(_15)]]
no
```

The derivation is given in the reverse order, because in the pre-compiled program I decided not to keep track of the derivation, but to use the list of previous goals instead, which is constructed in the reverse order.

Chapter 5

Conclusions

Until now different implementations of loop checking were presented. In this chapter, I'll compare the implementations and the loop checks themselves. Afterwards the question "How costly is loop checking" will be discussed.

5.1 The example programs

When a loop check is used, we have to make a choice between a weak loop check that prune late (or not at all) and a stronger loop check which prune earlier but is usually costlier (as shown later). The example program determines which is the best one. What does it means "the best one"? On one hand, the cheapest one is certainly the one that use no loop check at all. But on the other hand, it fails to detect loops, resulting in infinite computation, which is difficult to compare with a loop check that detects the loop and gives a finite result!

Our purpose is not to show where the different loop checks prune the SLD-tree , which is done in [BAK 89], but only their different cost. Thus, if two loop checks are compared, the object program and the initial goal is chosen such that the resulting SLD-tree is pruned, by the two loop checks, at the same place(s). In particular when a loop check is compared to the empty loop check, the object program does not loop.

In practice the transitive closure program presented in the example 1.1 is used with different relation r. One of the interest of this program is that it allows us to control the presence and length of loops easily by modifying the relation r.

The following numerical results are obtained using three different graph structures: one linear and two circular. The linear one (called *program 1* is the transitive closure program with the relation {r: r(a,c),..., r(y,z)} and the initial goal ←tc(a,z). The SLD-tree contains 79 nodes, one derivation that succeed and 27 that failed.

The first circular program (called *program 2*) is also the transitive closure program with the relation r:

```
r(a,b).  r(c,d).  r(e,f).  r(g,h).  r(j,i).  r(b,h).  r(h,j).  r(k,l).  r(m,n).
r(b,c).  r(d,e).  r(f,g).  r(i,h).  r(a,j).  r(a,h).  r(e,k).  r(l,m).  r(n,o).
r(o,p).  r(p,q).  r(q,r).  r(r,h).
```

and the initial goal ←tc(a,c). The SLD-tree produced by any of the full-comparison check have 96 nodes, one derivation that succeed, 29 that failed and five that are pruned. For single (respectively double) triangular check the number of nodes developed is of 110 (respectively 188), one (respectively one) derivation succeed, 34 (respectively 61) failed and 5 (respectively 5) are pruned.

The second circular program (called *program 3*) is a slightly different version of the transitive closure program [NS 91].

Let e be the edge relation of a digraph, and d be a subset of the nodes in the graph. Then the following program defines t to be the transitive closure restricted to paths starting at a node in d and such that all intermediate nodes have self-loops :

```
t(X,Y) :- d(X), e(W,W), e(W,Y), t(X,W).
t(X,Y) :- d(X), e(X,Y).
```

with the facts:

```
e(a,g).  e(g,g).  e(f,g).  e(f,f).  e(f,i).  e(i,i).  e(g,h).  e(c,h).  e(h,i).
e(l,a).  e(l,b).  e(b,a).  e(b,j).  e(d,c).  e(h,d).  e(e,k).  e(e,e).  e(i,e).
e(a,a).  e(b,b).  e(h,h).
d(a).    d(l).    d(e).    d(f).
```

and the initial goal ←t(X,k).

The SLD-tree produce by the equality loop checks (respectively subsumption loop checks) have 465 (respectively 433) nodes, 10 (respectively 10) derivation that succeed, 214 (respectively 130) that failed and 32 (respectively 116) that are pruned.

5.2 The different implementations

It appears that the most time- and space-consuming component of our implementations is the explicit manipulation of substitutions, which occurs both in the construction of the derivation (in the form of unification and application of substitution to the next goal) and in the loop check (in the form of matching and application of substitution to the resultant head). Consequently the cpu-time spent to find the resolvent of the current goal ("derive"-time) and the cpu-time spent in the loop checking (the cpu-time spent in the procedure check and to apply the substitution to the resultant head) ("check"-time) are good indications of the efficiency of the implementations.

Tables 5.1 and 5.2 show our measurements (in seconds of used cpu-time). In table 5.1 "program 1" with the initial goal ←tc(a,z) is interpreted for four different implementations and five loop checks (*non*, *cig*, *cir*, *sig*, *sir*).

program 1		meta	real-meta	real-metaWRV	pre
non	derive	78.0	65.5	4.1	0.1
	check	1.7	1.3	0	0
eig	derive	78.0	65.6	11.1	0.1
	check	17.2	16.7	15.0	14.4
eir	derive	78.2	65.8	11.2	0.1
	check	25.2	17.5	16.1	15.3
sig	derive	78.7	65.2	11.2	0.1
	check	26.7	23.5	23.0	21.8
sir	derive	78.4	64.8	11.2	0.1
	check	26.7	25.2	23.9	22.6

Table 5.1: The derive and check-time for interpreting *program 1* with different implementations

program 2		meta	real-meta	real-metaWRV	pre
eig	derive	93.2	84.1	14.6	0.3
	check	10.8	10.4	8.8	8.1
eir	derive	93.2	84.1	14.6	0.3
	check	11.6	11.2	9.5	8.7
sig	derive	93.6	84.4	14.5	0.3
	check	16.1	15.5	13.2	12.4
sir	derive	94.1	83.5	14.5	0.4
	check	16.7	15.7	13.8	12.9

Table 5.2: The derive and check-time for interpreting *program 2* with different implementations

Table 5.2 shows "program 2", with the initial goal ←tc(a,c), interpreted by the four implementations with four loop checks (*eig*, *eir*, *sig*, *sir*).

The four implementation are:

"*meta*": the first meta-interpreter presented in subsection 3.3.2;

"*real-meta*": the meta-interpreter that only said when a derivation succeed, as presented in subsection 3.3.4;

"*real-metaWRV*": the same meta-interpreter as "*real-meta*" except that it uses the PROLOG variables and not our "$ variables";

"*pre*": the pre-compiled program, presented in section 4.2, interpreted by the PROLOG system.

The first thing to note is that the "check"-time is almost the same for all the four implementations. There is a small difference between the two implementations that use the "$ variables" (*meta* and *real-meta*) and the two

program 1	empty	non	eig	evg	eir	evr
derive-time	65.5	65.6	65.6	65.7	65.8	65.4
check-time	1.3	1.3	16.7	17.7	17.5	19.1
comparison	0	0	1978	1978	1978	1978
literals in list	183	0	104	104	183	183
program 1	sig	svg	sir	svr	sir,st	sir,dt
derive-time	65.2	64.8	64.8	65.0	64.9	64.7
check-time	23.5	24.5	25.2	25.9	4.5	2.1
comparison	1978	1978	1978	1978	250	68
literals in list	104	104	183	183	183	35

Table 5.3: Comparision of the different loop checks for *program 1*

that use the PROLOG variables. This is because in the implementations with the PROLOG variables the application of substitution to the resultant head is explicit and so we can not measure it. This can be seen in table 5.1 for the loop check *non* where the "check"-time is not equal to zero for the first two meta-interpreters, this represent the time used to apply the substitution to the resultant head.

Now let us analyse the "derive"-time of the different implementations. The derive-time with *meta* is about 12 seconds bigger than with real-*meta* in table 5.1 and nine seconds in table 5.2. This represent the time used to find if a leaf of the SLD-tree failed and keep the derivation. In *real-meta* when a leaf failed it is the procedure find_new_goal_resultant which failed.

A large part of the time spent in the computation of the resolvent is used to apply the substitution to the new goal. This can be seen when we compare the derive-time of *real-meta* and of *real-metaWRV*, *derive-time* in *real-meta* is about six time *derive-time* in *real-metaWRV*. This is due to the fact that the application of substitution is made explicitly in *real-meta* and implicitly in *real-meta*WRV by the underlying PROLOG system which is very efficient.

The computation of the resolvent is even more efficient with the pre-compiled program, *pre*, because it is done completely implicitly by the underlying PROLOG system.

5.3 The different loop checks

In this section the different loop checks will be compared. To do this we use the result obtained by *real-meta* for the three programs. The results are displayed in table 5.3, 5.4 and 5.5. Where *derive-time* is the total cpu-time, in seconds, used to compute the next goal; *check-time* is the total cpu-time, in seconds, used to perform the loop checking itself. *Comparisons* is the total number of goal comparisons. *Literals in list* is the total number of literals stored in the list of goals and the list of resultant heads for the loop checking.

program 2	eig	evg	eir	evr		
derive-time	84.1	84.1	84.2	84.5		
check-time	10.4	11.1	11.2	11.5		
comparison	1150	1150	1150	1150		
literals in list	120	120	211	211		
program 2	sig	svg	sir	svr	sir,st	sir,dt
derive-time	84.4	83.9	83.5	82.9	95.3	167.3
check-time	15.5	15.5	15.7	15.8	4.5	5.2
comparison	1150	1150	1150	1150	245	174
literals in list	120	120	211	211	243	82

Table 5.4: Comparision of the different loop checks for *program 2*

program 3	eig	evg	eir	evr
derive-time	444.5	443.6	444.1	444.01
check-time	60.7	73.7	73.6	75.9
comparison	6832	6832	6832	6832
literals in list	879	879	1312	1312
program 3	sig	svg	sir	svr
derive-time	306.5	307.0	310.7	310.5
check-time	113.6	113.3	116.7	115.3
comparison	5472	5472	5472	5472
literals in list	647	647	964	964

Table 5.5: Comparision of the different loop checks for *program 3*

For the linear program, *program 1*, all the loop checks are used and for the two circular, *program 2* and *program 3*, the loop checks *empty* and *non* are not used since they do not detect loops and never give results.

For the first two programs, see table 5.3 and 5.4 the number of node in the SLD-tree developed is the same for all the full-comparison loop checks. Thus *derive-time* is the same for all the full-comparison loop checking. For the *program 3* the subsumption checks prune earlier than the equality checks, they develop respectively 562 and 678 nodes. So *derive-time* for the four subsumption checks is smaller than *derive-time* for the four equality checks.

The results of tables 5.3 and 5.4 show that the subsumption checks are significaly more expensive than the equality checks. Due to the small size of the goals derived with the initial goal ←tc(a,z) or ←tc(a,c) (one or two literals with a mean length of 1.5) the subsumption checks are approximately 40% more expensive.

In *program 3*, where the goals are bigger (from one to four literals with a mean length of 2.5), it is more difficult to evaluate the cost of subsumption checks because subsumption and equality checks do not develop the same

number of nodes in the SLD-tree. But it is possible to evaluate the time of one goal comparison (*check-time* divided by the number of goal comparisons done). The time for one goal comparison is about 0.01 second for the equality checks and 0.02 second for the subsumption checks. This means that the subsumption checks are about 100% more expensive.

We can see that if the mean size of the goals raises from 1.5 to 2.5 literals, the cost of the subsumption checks augments from 40% to 100%. If the mean size of the goals augments we can imagine that the cost of subsumption check increases in the same manner.

The result show that there is no much difference among the equality checks (and among the subsumption checks). The checks based on goals are slightly cheaper and use less space than those based on resultant. This represent the time use to compare the resultant heads and the space to store them.

The checks testing for instance are cheaper than those testing for variance. This is due to the implementation: first a substitution is computed, then it is tested if this substitution is renaming.

In table 5.3, the advantage of the triangular loop checks are evident: they need much less time. But this program does not show their disadvantage, they develop the same SLD-tree as the full-comparison checks and so make less comparison. Usually, they prune the derivation later, so they develop more nodes. This is shown in tables 5.4. They do not compare each goal, so they detect loops later than the full-comparison check and develop more node of the SLD-tree. Thus for the triangular checks *check-time* decreases but *derive-time* increases. Due to the fact that we use triangular numbers, the deeper we are in the SLD-tree the longer is the distance between two checks and we have to develop more nodes after the beginning of the loop to detect it.

If we look more precisely to the figures of *check-time* for the single and double triangular loop checks, we can see that single triangular check make more comparisons but *check-time* is smaller than for double triangular check. This is because the time to apply the substitution to the resultant head, which is done for each nodes, is counted in *check-time*. In table 5.3 the time to apply the substitution to one resultant head is about 0.016 second (*check-time* for empty loop check, 1.3 second, divided by the number of nodes, 76), therefore for the single (respectively double) triangular loop check that develop 109 nodes (respectively 187 nodes) the time used to apply substitution to resultant heads is about 1.7 seconds (respectively 3 seconds) and therefore the time for the comparison itself is 2.8 seconds (respectively 2.2).

5.4 How costly is the loop checking

To answer to the question "How costly is the loop checking?" we should compare an efficient PROLOG interpreter with loop checking with existing PROLOG interpreters. As developing a really efficient PROLOG interpreter

with loop checking involves a lot of work and it is not the purpose of this thesis, it will be helpful to use a meta-interpreter to have an idea of the result obtained with an efficient implementation.

But which implementation should we use for that purpose? This implementation is of course less efficient than the real one, but its inefficiency must be the same for the construction of the derivation and the loop checking itself.

The four implementations used in this chapter have almost the same loop checking efficiency. At the opposite, the construction of the derivation are quite different. In *real-metaWRV* and *pre*, due to the use of the PROLOG variables, the manipulation of the substitution is done implicitly and so is very efficient with respect of the loop checking (perhaps too efficient). In *real-meta* the application of substitution, unification and matching are made explicitly. Thus it seems that the construction of derivation and the loop checking are made with the same degrees of efficiency. In *meta*, the efficiency of the application of substitution, unification and matching is the same as for *real-meta*. But *meta* do more work than an usual PROLOG system: it stop when a goal failed and when a derivation is pruned.

I think *real-meta* is a good meta-interpreter to give an idea of the performance of a real PROLOG interpreter with loop check. The tables 5.3, 5.4 and 5.5 show the results obtain by those three programs interpreted by *real-meta*.

The cost of the loop check depends of course on the program and the check used. If the average size of the goals is bigger the time to compare goals will be bigger than for smaller goals. The three example use here have quite small goals. But one may suspect that it costs too much to apply the full-comparison loop check to a large example with few loops. In such cases, the use of a triangular loop checks definitely beats using no loop check at all. This applies in particular to programs that are still being tested/debugged: they are not supposed to loop but some loops may be present.

The construction of the derivation can be greatly improved with some common optimization techniques, such as last call optimization. It is conceivable that such optimization would increases the relative cost of the loop checking. On the other hand, our loop checking procedure itself could certainly be improved, for example by using some kind of "incremental" testing (an equality check tests first if two goals have the same length, then whether they have the same predicates in the same order and so on), ordering the previous goal by the probability to be similar to the current one. Also the storage and retrieval of previous goals could be improved by some hashing techniques. These optimizations would make loop checking less costly.

I think it's difficult to predict the cost of the loop checking with the implementations presented here. At the beginning the meta-interpreter was just seen as a prototype to show that it was possible to implement loop checking and to make some experiments to see what was the SLD-tree produced by some programs. It was only at the end of the work that the question "How costly is the loop checking?" was asked. This is why the different implementations do not use all the optimization techniques.

Anyway the loop checking always cost something, so an interpreter with loop checking is less efficient than one without. In some case where the efficiency is not too important, for example for debbuging programs or for writing programs in a more declarative sytle, the cost of loop checking is acceptable. The weakness of the loop checks presented here is the relatively small class of programs for which they are complete.

Appendix A

The meta-interpreter

This is the code of the PROLOG meta-interpreter, written in quintus PROLOG, describe in section 3.3.

It can only interpret pure PROLOG programs (without negation and cuts). It does not use ordinary variables, the variables must start with the character "$".

The user call

```
solvel(check,[a($X)]).
```

means solve the goal ←a(X) with the loop check check. The meta-interpreter and the program must be loaded in memory. In the program all the variables must be atoms that start with the character "$". For example a(X) :- b(Y) must be transformed in a($X) :- b($Y).

This implementation updates five counters:

two real counters: the "check-time" (the cpu-time spent in the procedure check and to apply the substitution to the resultant head) and the "derive-time" (the cpu-time spent to find the resolvent of the current goal).

three integer counter: the number of goal comparisons made, the number of node developed and the number of literals stored in the list of goals and the list of resultant heads for the loop checking.

```
/*#######################################################*/
/*                    Jean HENRARD    14 may 1991        */
/*                    CWI - Amsterdam - The Netherlands  */
/*                    FNDP - Namur - Belgium             */
/*#######################################################*/

/*#######################################################*/
/* The structure of solveI is :                          */
/*   solveI                                              */
/*        create_answer_substitution                     */
/*        print_derivation                               */
/*        solve                                          */
/*             check (cfr the definition of the check procedure) */
/*             find_new_goal_resultant                   */
/*                  apply_sub_sub                        */
/*                       apply_sub_expr                  */
/*                       member_sub                      */
/*                            append                     */
/*                       is_var                          */
/*                  apply_sub_expr                       */
/*                  append                               */
/*                  d_clause                             */
/*                       d_unif                          */
/*                            occur                      */
/*                                 is_var                */
/*                            append                     */
/*                            is_var                     */
/*                            replace_var_sub            */
/*                            apply_binding              */
/*                       transform_body_to_list          */
/*                       fresh_variables                 */
/*                            apply_sub_expr             */
/*                            make_fresh_substitution    */
/*                                 append                */
/*                            list_var                   */
/*#######################################################*/
```

```
/*=======================================================*/
/*  solveI(Check,Goal)                                   */
/*=======================================================*/
/*  type :  Check in { non, empty, evg, eig, evr, eir, svg, sig, svr, sir,  */
/*          (evg,st), (eig,st), (evr,st), (eir,st), (svg,st), (sig,st), (svr,st),  */
/*          (sir,st), (evg,dt), (eig,dt), (evr,dt), (eir,dt), (svg,dt), (sig,dt),  */
/*          (svr,dt), (sir,dt)}.                         */
/*          Goal = expression                            */
/*  relation - side effect : solve the goal 'Goal' with the loop check 'Check'  */
/*          if it finds a solution then prints the computed  answer  */
/*              solution, the depth where the solution is found and the  */
/*              derivation;                              */
/*          if it prunes the derivation then prints the depth  */
/*              where the branch is pruned and the derivation;  */
/*          if it finds a branch that fail then prints the depth  */
/*              where the branch fails and the derivation.  */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)  */
/*=======================================================*/

solveI(Check, Goal) :-
            /* Initialisation of the counters for                  */
        init_float(1),          /* check time                      */
        init_float(2),          /* derivation time                 */
        init_int(1),            /* number of goal comparisons      */
        init_int(2),            /* number of nodes developed       */
        init_int(3),            /* number of literals keep in the lists  */
    create_answer_substitution(Goal, [], Substitution),
    solve(Check, Goal, Goal, Substitution, [], [], 1, 0, 0, 0,Derivation),
            /* read of all the counters and display them           */
        read_float(1, TC),
        read_float(2, TD),
        read_int(1, GC),
        read_int(2, G),
        read_int(3, LL),
        nl, write('  check : '),
        write(Check),
        nl, write('derivation time '),
        write(TD),
        nl, write('time comparison '),
        write(TC),
        nl, write('number of goal comparisons '),
        write(GC),
        nl, write('number of node developed '),
        write(G),
        nl, write('number of literals in the lists '),
```

```
        write(LL),
    nl, write('derivation '),
    nl, print_derivation([Goal | Derivation]),
    nl, nl,
        fail.     /* for finding all the solutions              */
solveI(Check, _) :-
    /* no more solution, prints the final version of the counters*/
        read_float(1, TC),
        read_float(2, TD),
        read_int(1, GC),
        read_int(2, G),
        read_int(3, LL),
        nl, write('  check : '),
        write(Check),
        nl, write('derivation time '),
        write(TD),
        nl, write('time comparison '),
        write(TC),
        nl, write('number of goal comparisons '),
        write(GC),
        nl, write('number of node developed '),
        write(G),
        nl, write('number of literals in the lists '),
        write(LL).
```

```
/*==================================================================*/
/*  solve(Check, Goal, Resultant, Substitution, ListGoal,           */
/*        ListResult, LastVar, Depth, I, FI, Derivation)            */
/*==================================================================*/
/*  type :  Check in { non, empty, evg, eig, evr, eir, svg, sig, svr, sir,  */
/*          (evg,st), (eig,st), (evr,st), (eir,st), (svg,st), (sig,st), (svr,st),  */
/*          (sir,st), (evg,dt), (eig,dt), (evr,dt), (eir,dt), (svg,dt), (sig,dt),  */
/*          (svr,dt), (sir,dt)}.                                     */
/*          Goal, Resultant : expressions                            */
/*          Substitution : list of bindings                          */
/*          ListGoal, ListResult, Derivation : lists of expressions  */
/*          LastVar, Depth : integers                                */
/*  relation :                                                       */
/*  When the derivation is                                           */
/*          G0 => {C1,θ1} ... => Gk-1 => {Ck,θk}Gk                   */
/*  has been constructed, the meanings of the parameters are the following: */
/*          Goal = Gk (also used for loop checking);                 */
/*          Substitution = the list of bindings representing θ1...θk */
/*             restricted to the variables of G0;                    */
/*          LastVar = the number of variables used (needed for       */
/*             standardization apart.)                               */
/*  Input parameters for loop checking:                              */
/*          Check = the loop check that is used:                     */
/*          Resultant = G0S1...Sk;                                   */
/*          ListGoal = [Gk-1, ..., G0];                              */
/*          ListResult = [G0S1...Sk, ..., G0];                       */
/*          Depth = k.                                               */
/*          FI = 1/2 I(I + 1) and 1/2(I - 1)- I < D ≤ 1/2 i (i + 1)  */
/*             if D = FI  then D is a triangular number              */
/*  If a double triangular check is used, ListGoal (ListResult)      */
/*  contains only the goals (resultant heads) with a triangular      */
/*  index.  When "non" is used, these lists are not maintained.      */
/*  LisResult is only maintain for loop checks for resultant         */
/*  or "empty" loop check.                                           */
/*  Output (provided that a finite derivation is generated)          */
/*          Derivation = [Gk+1, ..., Gn, true] means Gn is empty     */
/*                       [Gk+1, ..., Gn, prune] means Gn is pruned   */
/*                       [Gk+1, ..., Gn, false] means Gn is failed   */
/*==================================================================*/
```

```
    /* if a solution is found (the goal to be solved is empty)          */
solve(Check, [], Resultant, Substitution, ListGoal, ListResult,
          LastVar, Depth, I, FI, [true]) :-
     nl, write(Substitution),
     nl, write('depth '),
     write(Depth).
    /* if Literal is a system predicate                                 */
solve(Check,  [Literal | Goal], Resultant, Substitution, ListGoal,
          ListResult, LastVar, Depth, I, FI, [NewGoal | Derivation]) :-
     syst(Literal),
     !,
     call_syst(Literal, CAS),
     apply_sub_expr(Goal, CAS, NewGoal),
     apply_sub_expr(Resultant, CAS, NewResultant),
     apply_sub_sub(Substitution, CAS, NewSubstitution),
     update_depth(Depth, I, FI, NewDepth, NewI, NewFI),
     solve(Check, NewGoal, NewResultant, NewSubstitution,
          [NewGoal | ListGoal], [NewResultant | ListResult],
          LastVar, NewDepth, NewI, NewFI, Derivation).
    /* we perform the loop check (Check) to see if we can continue,     */
    /* then we compute the next goal to be solved and the               */
    /* substitution (NewSubstitution) and the new resultant             */
    /* (NewResultant) and solve the new goal                            */
solve(Check, Goal, Resultant, Substitution, ListGoal, ListResult,
          LastVar, Depth, I, FI, Derivation) :-
           /* Goal is not empty and ists first literal is not a         */
           /* built-in predicate                                        */
     check(Check, Goal, Resultant, ListGoal, ListResult,
          Depth, FI, NewListGoal, NewListResult),
     !,
     find_new_goal_resultant(Goal, Resultant, Substitution, LastVar,
          NewGoal, NewResultant, NewSubstitution, NewLastVar),
     ( NewGoal = [false|_] ->
               /* the current goal has no resolvent                     */
          Derivation = [false],
          nl, write('detph '),
          write(Depth)
     ;         /* the current goal has a resolvent                     */
          update_depth(Depth, I, FI, NewDepth, NewI, NewFI),
          Derivation = [NewGoal | NewDerivation],
          solve(Check, NewGoal, NewResultant, NewSubstitution,
               NewListGoal, NewListResult, NewLastVar, NewDepth,
               NewI, NewFI, Derivation)
     ).
```

```
    /* if we are in a infinit loop (not(check( ... ))) then we          */
    /* prune the derivation at this point.                              */
solve(Check, [A | Goal], Resultant, Substitution, ListGoal, ListResult,
          LastVar, Depth, I, FI, [prune]) :-
           /* not(check( ... )), a loop is detected                     */
     nl, write('depth '),
     write(Depth).


/* definition of the built-in predicate                                 */

syst(_ = _).
syst(_ >= _).
syst(_ > _).
syst(_ < _).
syst(_ =< _).
syst(_ is _).

call_syst(X=Y,  Substitution) :-
     d_unif([X], [Y], Substitution).
call_syst(X>=Y, []) :-
     X >= Y.
call_syst(X>Y, []) :-
     X > Y.
call_syst(X<Y, []) :-
     X < Y.
call_syst(X=<Y, []) :-
     X =< Y.
call_syst(X is Y, [eq(X, Z)]) :-
     is_var(X),
     !,
     Z is Y.
call_syst(X is Y, []) :-
     X is Y.
```

```
/*==================================================================*/
/*  create_answer_substitution(Expr, OldSubstitution,               */
/*          NewSubstitution)                                        */
/*==================================================================*/
/*  type :  Expr : expression                                       */
/*          OldSubstitution, NewSubstitution : lists of binding     */
/*  relation : if [$X1, ... ,$Xn] is the list of variables          */
/*              occuring in Expr then NewSubstitution               */
/*              = [eq($Xi1, $Xi1),... ,eq($Xik, $Xik) | OldSubstitution]  */
/*               where the $Xij are the variables that does         */
/*              not appear in OldSubstitution                       */
/*  directionnality : in(gr, gr, var) : out(gr, gr, gr)             */
/*==================================================================*/

create_answer_substitution([], OldSubstitution, OldSubstitution).
create_answer_substitution([E | Expr], OldSubstitution, NewSubstitution) :-
        is_var(E),
        !,
        (member_sub(_, OldSubstitution, E) ->
                create_answer_substitution(Expr, OldSubstitution, NewSubstitution)
        ;       create_answer_substitution(Expr, [eq(E, E) | OldSubstitution],
                NewSubstitution)
        ).
create_answer_substitution([E | Expr], OldSubstitution, NewSubstitution) :-
                /* not (is_var(E)) */
        E =.. [_ | List],
        append(List, Expr, NewExpr),
        create_answer_substitution(NewExpr, OldSubstitution, NewSubstitution).
```

```
/*------------------------------------------------------------------*/
/*  print_derivation(Derivation)                                    */
/*------------------------------------------------------------------*/
/*  type :  Derivation : a list                                     */
/*  relation - side effect :                                        */
/*          Derivation = [L1, ... ,Ln]                              */
/*          then it prints                                          */
/*              L1                                                  */
/*              ....                                                */
/*              Ln                                                  */
/*  directionnality : in(novar) : out(novar)                        */
/*------------------------------------------------------------------*/

print_derivation([]).
print_derivation([Goal | Derivation]) :-
        write(Goal),
        nl,
        print_derivation(Derivation).


/*------------------------------------------------------------------*/
/*  update_depth(Depth, I, FI, NewDepth, NewI, NewFI)               */
/*------------------------------------------------------------------*/
/*  type :  Depth, I, FI, NewDepth, NewI, NewFI : integers          */
/*  relation :                                                      */
/*          if Depth = FI                                           */
/*          then NewI = I + 1                                       */
/*              NewFI = FI + NewI                                   */
/*              NewDepth = Depth + 1                                */
/*          else NewI = I                                           */
/*              NewFI = FI                                          */
/*              NewDepth = Depth + 1                                */
/*  directionnality : in(gr, gr, gr, var, var, var)                 */
/*                    out(gr, gr, gr, gr, gr, gr)                   */
/*------------------------------------------------------------------*/

update_depth(Depth, I, Depth, NewDepth, NewI, NewFI) :-
        !,
        NewI is I + 1,
        NewFI is Depth + NewI,
        NewDepth is Depth + 1.
update_depth(Depth, I, FI, NewDepth, I, FI) :-
        NewDepth is Depth + 1.
```

```
/*#######################################################*/
/*                THE LOOP CHECK PROCEDURES              */
/*#######################################################*/


/*=======================================================*/
/*  check(Check, Goal, Resultant, ListGoal, ListResult, Depth,  */
/*        NewListGoal, NewListResult)                    */
/*=======================================================*/
/*  check use  CASE 1  : /////                           */
/*             CASE 2  : check_EVG                       */
/*                       renaming                        */
/*             CASE 3  : check_EIG                       */
/*                       compute_substitution            */
/*             CASE 4  : check_EVR                       */
/*                       same_EVR                        */
/*                            renaming                   */
/*                            same_substitution          */
/*             CASE 5  : check_EIR                       */
/*                          same_EIR                     */
/*                            compute_substitution       */
/*                            same_substitution          */
/*             CASE 6  : check_SVG                       */
/*                          incl_ren                     */
/*             CASE 7  : check_SIG                       */
/*                          incl_sub                     */
/*             CASE 8  : check_SVR                       */
/*                          same_SVR                     */
/*                            incl_ren                   */
/*                            renaming                   */
/*                            same_substitution          */
/*             CASE 9  : check_SIR                       */
/*                          same_SIR                     */
/*                            incl_sub                   */
/*                            compute_substitution       */
/*                            same_substitution          */
/*             CASE 10 : check_t                         */
/*                          check_EVG, check_EIG, check_EVR,  */
/*                          check_EIR, check_SVG, check_EIG,  */
/*                          check_SVR, check_SIR         */
/*             CASE 11 : check_t                         */
/*                          check_EVG, check_EIG, check_EVR,  */
/*                          check_EIR, check_SVG, check_EIG,  */
/*                          check_SVR, check_SIR         */
/*=======================================================*/
```

```
/*=======================================================*/
/*  type :  Check in { non, empty, evg, eig, evr, eir, svg, sig, svr, sir,  */
/*          (evg,st), (eig,st), (evr,st), (eir,st), (svg,st), (sig,st), (svr,st),  */
/*          (sir,st), (evg,dt), (eig,dt), (evr,dt), (eir,dt), (svg,dt), (sig,dt),  */
/*          (svr,dt), (sir,dt)}.                         */
/*          Goal, Resultant : expressions = lists of terms and/or atoms  */
/*          ListGoal, ListResult, NewListGoal, NewListResult :  */
/*               lists of expressions                    */
/*          depth : integer                              */
/*  relation : Check is the kind of loop check to be performed  */
/*          Depth is the depth where the loop check occur  */
/*               ListGoal = [G1, .. ,Gn]                 */
/*               ListResult = [R1, .. ,Rn]               */
/*          if  there is no i such that Goal 'is sufficiently similar'  */
/*               to Gi w.r.t. Check                      */
/*               and Resultant 'is sufficiently similar' to Ri w.r.t. Check  */
/*          then NewListGoal = [Goal | ListGoal]         */
/*               NewListResult = [resultant | ListResult]  */
/*          exept if Check = ***_d_t then                */
/*               if Depth in {1/2 i(i + 1) |  i in N}    */
/*               then NewListGoal = [Goal | ListGoal]    */
/*                    NewListResult = [Resultant | ListResult]  */
/*               else NewListGoal = ListGoal             */
/*                    NewListResult = ListResult         */
/*  directionnality :                                    */
/*          in(gr, gr, gr, gr, gr, gr, var, var) :       */
/*          out(gr, gr, gr, gr, gr, gr, gr, gr)          */
/*=======================================================*/

/*******************************************************\
CASE 1          no loop check
\*******************************************************/

check(empty, Goal, Resultant, ListGoal, ListResult, Depth, FI, [Goal | ListGoal],
            [Resultant | ListResult]) :-
          length(Goal, G),
          length(Resultant, R),
          N is G + R,
          add_int(3, N).

check(non, Goal, Resultant, ListGoal, ListResult, Depth, FI, ListGoal, ListResult).
```

```
/*************************************************************\
CASE 2          Equals Variant of Goal
\*************************************************************/

check(evg, Goal, Resultant, ListGoal, ListResult, Depth, FI, [Goal | ListGoal],
                ListResult):-
      check_EVG(Goal, ListGoal),
            length(Goal, G),
            add_int(3, G).


/*-----------------------------------------------------------------------------*/
/*  check_EVG(Goal, ListGoal)                                                  */
/*-----------------------------------------------------------------------------*/
/*  type :  Goal : expression                                                  */
/*          ListGoal : list of expressions = [G1, .. ,Gn]                      */
/*  relation : there is no i and renaming θ  such that Gi θ = Goal             */
/*  directionnality : in(gr, gr) :  out(gr, gr)                                */
/*-----------------------------------------------------------------------------*/

check_EVG(Goal, []).
check_EVG(Goal, [G | ListGoal]):-
            add_int(1, 1),
            statistics(runtime,_),
      \+ renaming(G, Goal, Renaming),
            statistics(runtime, [_ , T]),
            add_float(1, T),
      check_EVG(Goal, ListGoal).


/*************************************************************\
CASE 3          Equals Instance of Goal
\*************************************************************/

check(eig, Goal, Resultant, ListGoal, ListResult, Depth, FI, [Goal | ListGoal],
                ListResult):-
      check_EIG(Goal, ListGoal),
            length(Goal, G),
            add_int(3, G).
```

```
/*-----------------------------------------------------------------------------*/
/*  check_EIG(Goal, ListGoal)                                                  */
/*-----------------------------------------------------------------------------*/
/*  type :  Goal : expression                                                  */
/*          ListGoal : list of expressions = [G_1, .. ,G_n]                    */
/*  relation : there is no i and substitution θ  such that Gi θ = Goal         */
/*  directionnality : in(gr, gr) :  out(gr, gr)                                */
/*-----------------------------------------------------------------------------*/

check_EIG(Goal, []).
check_EIG(Goal, [G | ListGoal]):-
            add_int(1, 1),
            statistics(runtime, _),
      \+ compute_substitution(G, Goal, Substitution),
            statistics(runtime, [_ , T]),
            add_float(1, T),
      check_EIG(Goal, ListGoal).


/*************************************************************\
CASE 4          Equals Variant of Resultant
\*************************************************************/

check(evr, Goal, Resultant, ListGoal, ListResult, Depth, FI, [Goal | ListGoal],
                [Resultant | ListResult]):-
      check_EVR(Goal, Resultant, ListGoal, ListResult),
            length(Goal, G),
            length(Resultant, R),
            N is G + R,
            add_int(3, N).


/*-----------------------------------------------------------------------------*/
/*  check_EVR(Goal, Resultant, ListGoal, ListResult)                           */
/*-----------------------------------------------------------------------------*/
/*  type :  Goal, Resultant : expressions                                      */
/*          ListGoal = list of expressions = [G1, .. ,Gn]                      */
/*          ListResult = list of expressions = [R1, .. ,Rn]                    */
/*  relation : there is no i and renaming θ  such that Gi θ = Goal             */
/*             and Ri θ = Resultant                                            */
/*  directionnality : in(gr, gr) :  out(gr, gr)                                */
/*-----------------------------------------------------------------------------*/
```

```
check_EVR(Goal, Resultant, [], []).
check_EVR(Goal, Resultant, [G | ListGoal], [R | ListResult]):-
        add_int(1, 1),
        statistics(runtime, _),
    \+ same_EVR(Goal, Resultant, G, R),
        statistics(runtime, [_ , T]),
        add_float(1, T),
    check_EVR(Goal, Resultant, ListGoal, ListResult).


/*-----------------------------------------------------------------------*/
/*  same_EVR(Goal, Resultant, G, R)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant, G, R : expressions                          */
/*  relation : there is a renaming θ  such that G θ = Goal               */
/*             and R θ = Resultant                                       */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

same_EVR(Goal, Resultant, G, R):-
    renaming(G, Goal, Renaming),
    renaming(R, Resultant, Renaming1),
    same_substitution(Renaming, Renaming1).


/*********************************************************************\
CASE 5        Equals Instance of Resultant
\*********************************************************************/

check(eir, Goal, Resultant, ListGoal, ListResult, Depth, FI, [Goal | ListGoal],
            [Resultant | ListResult]):-
    check_EIR(Goal, Resultant, ListGoal, ListResult),
        length(Goal, G),
        length(Resultant, R),
        N is G + R,
        add_int(3, N).
```

```
/*-----------------------------------------------------------------------*/
/*  check_EIR(Goal, Resultant, ListGoal, ListResult)                     */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant : expressions                                */
/*          ListGoal = list of expressions = [G1, .. ,Gn]                */
/*          ListResult = list of expressions = [R1, .. ,Rn]              */
/*  relation : there is no i and substitution θ  such that Gi θ = Goal   */
/*             and Ri θ = Resultant                                      */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

check_EIR(Goal, Resultant, [], []).
check_EIR(Goal, Resultant, [G | ListGoal], [R | ListResult]):-
        add_int(1, 1),
        statistics(runtime, _),
    \+ same_EIR(Goal, Resultant,G, R),
        statistics(runtime, [_ , T]),
        add_float(1, T),
    check_EIR(Goal, Resultant, ListGoal, ListResult).


/*-----------------------------------------------------------------------*/
/*  same_EIR(Goal, Resultant, G, R)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant, G, R : expressions                          */
/*  relation : there is a substitution θ  such that G θ = Goal           */
/*             and R θ = Resultant                                       */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

same_EIR(Goal, Resultant, G, R):-
    compute_substitution(G, Goal, SubstitutionGoal),
    compute_substitution(R, Resultant, SubstitutionResult),
    same_substitution(SubstitutionGoal, SubstitutionResult).
```

```
/*************************************************************\
CASE 6          Subsumes Variant of Goal
/*************************************************************/

check(svg, Goal, Resultant, ListGoal, ListResult, Depth, FI, [Goal | ListGoal],
                ListResult):-
     check_SVG(Goal, ListGoal),
          length(Goal, G),
          add_int(3, G).


/*-----------------------------------------------------------------------------*/
/*   check_SVG(Goal, ListGoal)                                                 */
/*-----------------------------------------------------------------------------*/
/*   type :  Goal : expression                                                 */
/*           ListGoal : list expressions = [G1, .. ,Gn]                        */
/*   relation : there is no i and renaming θ such that Gi θ = Goal             */
/*   directionnality : in(gr, gr) :  out(gr, gr)                               */
/*-----------------------------------------------------------------------------*/

check_SVG(Goal, []).
check_SVG(Goal, [G | ListGoal]):-
          add_int(1, 1),
          statistics(runtime, _),
     \+ incl_ren(G, Goal, Renaming),
          statistics(runtime, [_ , T]),
          add_float(1, T),
     check_SVG(Goal, ListGoal).


/*************************************************************\
CASE 7          Subsumes Instance of Goal
\*************************************************************/

check(sig, Goal, Resultant, ListGoal, ListResult, Depth, FI, [Goal | ListGoal],
                ListResult):-
     check_SIG(Goal, ListGoal),
          length(Goal, G),
          add_int(3, G).
```

```
/*-----------------------------------------------------------------------------*/
/*   check_SIG(Goal, ListGoal)                                                 */
/*-----------------------------------------------------------------------------*/
/*   type :  Goal : expression                                                 */
/*   ListGoal : list expressions = [G1, .. ,Gn]                                */
/*   relation : there is no i and substitution θ                               */
/*              such that Gi θ = Goal                                          */
/*   directionnality : in(gr, gr) :  out(gr, gr)                               */
/*-----------------------------------------------------------------------------*/

check_SIG(Goal, []).
check_SIG(Goal, [G | ListGoal]):-
          add_int(1, 1),
          statistics(runtime, _),
     \+ incl_sub(G, Goal, Substitution),
          statistics(runtime, [_ , T]),
          add_float(1, T),
     check_SIG(Goal, ListGoal).


/*************************************************************\
CASE 8          Subsumes Variant of Resultant
\*************************************************************/

check(svr, Goal, Resultant, ListGoal, ListResult, Depth, FI, [Goal | ListGoal],
                [Resultant | ListResult]):-
     check_SVR(Goal, Resultant, ListGoal, ListResult),
     length(Goal, G),
     length(Resultant, R),
     N is G + R,
     add_int(3, N).


/*-----------------------------------------------------------------------------*/
/*   check_SVR(Goal, Resultant, ListGoal, ListResult)                          */
/*-----------------------------------------------------------------------------*/
/*   type :  Goal, Resultant : expressions                                     */
/*           ListGoal = list of expressions = [G1, .. ,Gn]                     */
/*           ListResutl = list of expressions = [R1, .. ,Rn]                   */
/*   relation : there is no i and renaming θ such that Gi θ include in Goal    */
/*              and Ri θ = Resultant                                           */
/*   directionnality : in (gr, gr, gr, gr) : out (gr, gr, gr, gr)              */
/*-----------------------------------------------------------------------------*/
```

```
check_SVR(Goal, Resultant, [], []).
check_SVR(Goal, Resultant, [G | ListGoal], [R | ListResult]):-
        add_int(1, 1),
        statistics(runtime, _),
    \+ same_SVR(Goal, Resultant, G, R),
        statistics(runtime, [_ , T]),
        add_float(1, T),
    check_SVR(Goal, Resultant, ListGoal, ListResult).


/*-----------------------------------------------------------------------*/
/*  same_SVR(Goal, Resultant, G, R)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant, G, R : expressions                          */
/*  relation : there is a renaming θ such that G θ include in Goal       */
/*             and R θ = Resultant                                       */
/*  directionnality : in (gr, gr, gr, gr) : out (gr, gr, gr, gr)         */
/*-----------------------------------------------------------------------*/

same_SVR(Goal, Resultant, G, R):-
    incl_ren(G, Goal, RenamingGoal),
    renaming(R. Resultant, RenamingResultant),
    same_substitution(RenamingResultant, RenamingGoal).


/*********************************************************************\
CASE 9        Subsumes Instance of Resultant
\*********************************************************************/

check(sir, Goal, Resultant, ListGoal, ListResult, Depth, FI, [Goal | ListGoal],
        [Resultant | ListResult]):-
    check_SIR(Goal, Resultant, ListGoal, ListResult),
        length(Goal, G),
        length(Resultant, R),
        N is G + R,
        add_int(3, N).
```

```
/*-----------------------------------------------------------------------*/
/*  check_SIR(Goal, Resultant, ListGoal, ListResult)                     */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant : expressions                                */
/*          ListGoal = list of expressions = [G1, .. ,Gn]                */
/*          ListResult = list of expressions = [R1, .. ,Rn]              */
/*  relation : there is no i and substitution θ such that Gi θ include in Goal */
/*             and Ri θ = Resultant                                      */
/*  directionnality : in (gr, gr, gr, gr) : out (gr, gr, gr, gr)         */
/*-----------------------------------------------------------------------*/

check_SIR(Goal, Resultant, [], []).
check_SIR(Goal, Resultant, [G | ListGoal], [R | ListResult]):-
        add_int(1, 1),
        statistics(runtime, _),
    \+ same_SIR(Goal, Resultant, G, R),
        statistics(runtime, [_ , T]),
        dd_float(1, T),
    check_SIR(Goal, Resultant, ListGoal, ListResult).


/*-----------------------------------------------------------------------*/
/*  same_SIR(Goal, Resultant, G, R)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant, G, R : expressions                          */
/*  relation : there is a substitution θ such that G . θ include in Goal */
/*             and R . θ = Resultant                                     */
/*  directionnality : in (gr, gr, gr, gr) : out (gr, gr, gr, gr)         */
/*-----------------------------------------------------------------------*/

same_SIR(Goal, Resultant, G, R):-
    incl_sub(G, Goal, SubstitutionGoal),
    compute_substitution(R, Resultant, SubstitutionResultant),
    same_substitution(SubstitutionResultant, SubstitutionGoal).
```

```
/*************************************************************\
CASE 10          Single Triangle loop checks
\*************************************************************/

check((Full, st), Goal, Resultant, ListGoal, ListResult, Depth, Depth,
                [Goal | ListGoal], [Resultant | ListResult]):-
      !,
      check_t(Full, Goal, Resultant, ListGoal, ListResult),
            length(Goal, G),
            length(Resultant, R),
            N is G + R,
            add_int(3, N).
check((Full, st), Goal, Resultant, ListGoal, ListResult, Depth, FI,
                [Goal | ListGoal], [Resultant | ListResult]):-
            length(Goal, G),
            length(Resultant, R),
            N is G + R,
            add_int(3, N).


/*************************************************************\
CASE 11          Double Triangle loop checks
\*************************************************************/

check((Full, dt), Goal, Resultant, ListGoal, ListResult, Depth, Depth,
                [Goal|ListGoal],[Resultant|ListResult]) :-
      !,
      check_t(Full, Goal, Resultant, ListGoal, ListResult),
            length(Goal, G),
            length(Resultant, R),
            N is G + R,
            add_int(3, N).
check((Full, dt), Goal, Resultant, ListGoal, ListResult, Depth, FI,
                ListGoal, ListResult).
```

```
/*-----------------------------------------------------------------------*/
/*  check_t(Check, Goal, Resultant, ListGoal, ListResult)                */
/*-----------------------------------------------------------------------*/
/*  type :  Check in {non, evg, eig,evr,eir,svg,sig,svr,sir}             */
/*          Goal,  Resultant : expressions                               */
/*          ListGoal, ListResult : lists of expressions                  */
/*  relation : ListGoal = [G1, .. ,Gn]                                   */
/*          ListResult = [R1, .. ,Rn]                                    */
/*          there is no i such that Goal, Resultant is sufficiently      */
/*              similar to Gi,Ri w.r.t. Check                            */
/*  directionnality : in (gr, novar, novar, novar, novar)                */
/*              : out (gr, novar, novar, novar, novar)                   */
/*-----------------------------------------------------------------------*/

check_t(evg, Goal, Resultant, ListGoal, ListResult):-
      check_EVG(Goal, ListGoal).
check_t(eig, Goal, Resultant, ListGoal, ListResult):-
      check_EIG(Goal, ListGoal).
check_t(evr, Goal, Resultant, ListGoal, ListResult):-
      check_EVR(Goal, Resultant, ListGoal, ListResult).
check_t(eir, Goal, Resultant, ListGoal, ListResult):-
      check_EIR(Goal, Resultant, ListGoal, ListResult).
check_t(svg, Goal, Resultant, ListGoal, ListResult):-
      check_SVG(Goal, ListGoal).
check_t(sig, Goal, Resultant, ListGoal, ListResult):-
      check_SIG(Goal, ListGoal).
check_t(svr, Goal, Resultant, ListGoal, ListResult):-
      check_SVR(Goal, Resultant, ListGoal, ListResult).
check_t(sir, Goal, Resultant, ListGoal, ListResult):-
      check_SIR(Goal, Resultant, ListGoal, ListResult).
```

```
/*#####################################################*/
/*                FIND THE NEXT GOAL TO SOLVE                */
/*#####################################################*/


/*-----------------------------------------------------------------------------*/
/*  find_new_goal_resultant(PrevGoal, PrevResultant, Substitution,            */
/*          LastVar, NewGoal, NewResultant, NewSubstitution,                  */
/*          NewLastVar)                                                       */
/*-----------------------------------------------------------------------------*/
/*  type :  PrevGoal, PrevResultant, NewGoal, NewResultant : expressions      */
/*          Substitution, NewSubstitution : lists of bindings                 */
/*          LastVar, NewLastVar : integers                                    */
/*  relation : NewGoal is PrevGoal where we replace the first  litteral is    */
/*              replaced by its body (Body) if it exist else by the atom 'false' */
/*          NewResultant is PrevResultant to which we apply the               */
/*              unification computes to found Body                            */
/*          The new variables are numered from LastVar to NewLastVar.         */
/*  directionnality : in(gr,  gr, gr, gr, var, var, var, var)                 */
/*              out(gr, gr, gr, gr, gr, gr, gr, gr)                           */
/*-----------------------------------------------------------------------------*/

find_new_goal_resultant([A | Goal],PrevResultant, Substitution, LastVar,
                NewGoal, NewResultant, NewSubstitution, NewLastVar) :-
        statistics(runtime, _),
    d_clause(A, Goal, NewGoal, SubstitutionClause, LastVar, NewLastVar),
    apply_sub_sub(Substitution, SubstitutionClause, NewSubstitution),
        statistics(runtime, [_ , T]),
        add_float(2, T),
        statistics(runtime, _),
    apply_sub_expr(PrevResultant, SubstitutionClause, NewResultant),
        statistics(runtime, [_ , T1]),
        add_float(1, T1),
        add_int(2, 1).
```

```
/*-----------------------------------------------------------------------------*/
/*  d_clause(Head, Goal, NewGoal, Unifier, Number, NewNumber)                 */
/*-----------------------------------------------------------------------------*/
/*  type :  Head : f(t1, .. ,tn)                                              */
/*              ti = term                                                     */
/*          Goal, NewGoal : expressions                                       */
/*          Unifier : list of bindings                                        */
/*          Number, NewNumber : integers                                      */
/*  relation : if there is a clause in the program (H :- B)                   */
/*              and an unifier Unifier such that                              */
/*              Head . Unifier = H . Unifier                                  */
/*          then Body = B . Unifier                                           */
/*              and all the variables that appear in H and B are              */
/*              renamed, NewNumber - Number is the number of                  */
/*              variables renamed                                             */
/*          else NewGoal = [false | Goal], Unifier = [], NewNumber = Number   */
/*  directionnality : in (gr, gr, var, var, gr, var                           */
/*                  out (gr, gr, gr, gr, gr, gr)                              */
/*-----------------------------------------------------------------------------*/

d_clause(Head, Body, Unifier, Number, NewNumber) :-
    functor(Head, F, N),
        /* This part checks only if there is at least a                      *\
         * clause which head unify with Head, it can be removed              *
         * as the second clause of d_clause if we are not                    *
         * interested in the fact that the interpreter stops                 *
        \* when it finds a failing branch                                    */
        functor(NewHead1, F, N),
        clause(NewHead1, B11),
        d_unif([NewHead1], [Head], Unifier1),
        !,
    functor(NewHead, F, N),
    clause(NewHead, B),
    fresh_variables([NewHead | [B]], [H | [B1]], Number, NewNumber),
    append_body_to_goal(B1, Goal, Goal1),
    d_unif([H], [Head], Unifier),
    apply_sub_expr(Goal1, Unifier, NewGoal).
d_clause(Head,[false], [], Number, Number).
        /* if there is no clause head unifying with Head                     */
```

```
/*-----------------------------------------------------------------------*/
/*  fresh_variables(In, Out, Number, NewNumber)                          */
/*-----------------------------------------------------------------------*/
/*  type :  in, out : expressions                                        */
/*          Number, Newnumber : integer                                  */
/*  relation : Out is In where we change all the name of the variables to */
/*             unused ones                                               */
/*          NewNumber - Number is the number of renamed variables        */
/*  directionnality : in (gr, var, gr, var) : out (gr, gr, gr, gr)       */
/*-----------------------------------------------------------------------*/
/*  ex :    input  In = [f(SX,SY),p(SX,SZ),g(SY)]                        */
/*             Number = 1                                                */
/*          output  Out = [f(SX1,SY2),p(SX1,SZ3),g(SY2)]                 */
/*             NewNumber = 4                                             */
/*-----------------------------------------------------------------------*/

fresh_variables(In, Out, Number, NewNumber) :-
      list_var(In, [], List_of_var_In),
      make_fresh_substitution(List_of_var_In, Substitution, Number,
            NewNumber),
      apply_sub_expr(In, Substitution, Out).


/*-----------------------------------------------------------------------*/
/*  append_body_to_goal(Body, Goal, NewGoal)                             */
/*-----------------------------------------------------------------------*/
/*  type :  Body is the body like you recieve it from  clause(Head, Body) */
/*          Goal, NewGoal : expressions                                  */
/*  relation : NewGoal = [Body transformed in a list | Goal]             */
/*-----------------------------------------------------------------------*/

append_body_to_goal(true, Goal, Goal) :-
    !.
append_body_to_goal(((A,B), Goal, [A|B_Goal]) :-
    !,
    append_body_to_goal(B, Goal, B_Goal).
append_body_to_goal((B, Goal, [B|Goal]).
```

```
/*#######################################################*/
/*  COMPUTE SUBSTITUTION, RENAMING, UNIFICATION, ...     */
/*#######################################################*/

/*-----------------------------------------------------------------------*/
/*  compute_substitution(Left, Right, Substitution)                      */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of terms and/or atoms      */
/*          Substitution : list of bindings                              */
/*  relation : there is a substitution Substitution such that            */
/*             Left . Substitution = Right                               */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)                */
/*-----------------------------------------------------------------------*/

compute_substitution(Left, Right, Substitution) :-
      compute_substitution_1(Left, Right, [], Substitution).


/*-----------------------------------------------------------------------*/
/*  compute_substitution_1(Left, Right, FirstSubstitution,               */
/*          Substitution)                                                */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of terms and/or atoms      */
/*          FirstSubstitution, Substitution : list of binding s          */
/*  relation : there is a substitution θ such that Left . θ = Right      */
/*          Substitution = [θ | FirstSubstitution]                       */
/*             without duplicates                                        */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)        */
/*-----------------------------------------------------------------------*/

compute_substitution_1([], [], FirstSubstitution, FirstSubstitution) :-
      !.
compute_substitution_1([L | Left], [R | Right], FirstSubstitution,
            Substitution) :-
      is_var(L),
      !,
      ( member_sub(L, FirstSubstitution, Term) ->
      R = Term,
      compute_substitution_1(Left, Right, FirstSubstitution, Substitution)
      ;     compute_substitution_1(Left, Right, [eq(L, R) | FirstSubstitution],
            Substitution)
      ).
compute_substitution_1([L | Left], [R | Right], FirstSubstitution,
            Substitution) :-
      !,
```

```
    functor(L, F, N),
    functor(R, F, N),
    L =.. [F | ListL],
    append(ListL, Left, NewLeft),
    R =.. [F | ListR],
    append(ListR, Right, NewRight),
    compute_substitution_1(NewLeft, NewRight, FirstSubstitution,
        Substitution).


/*-----------------------------------------------------------------*/
/*  renaming(Left, Right, Renaming)                                */
/*-----------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of terms and/or atoms */
/*          Renaming : list of bindings                             */
/*  relation : there is a renaming Renaming such that               */
/*             Left . Renaming = Right                              */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)           */
/*-----------------------------------------------------------------*/

renaming(Left, Right, Renaming) :-
    compute_substitution(Left, Right, Renaming),
    is_renaming(Renaming).


/*-----------------------------------------------------------------*/
/*  is_renaming(Renaming)                                          */
/*-----------------------------------------------------------------*/
/*  type :  Renaming : list of bindings                            */
/*  relation : Renaming = [eq(V1, T1), .. ,eq(Vn, Tn)]             */
/*          T1, .. ,Tn are distinct variables                      */
/*  directionnality : in(gr) : out(gr)                             */
/*-----------------------------------------------------------------*/

is_renaming([]).
is_renaming([eq(_ ,T) | Renaming]) :-
    is_var(T),
    no_in_renaming(T, Renaming),
    is_renaming(Renaming).


/*-----------------------------------------------------------------*/
/*  no_in_renaming(T, Renaming)                                    */
/*-----------------------------------------------------------------*/
/*  type :  T : a variable                                         */
/*          Renaming : list of bindings                            */
/*  relation : Renaming = [eq(V1, T1), .. ,eq(Vn, Tn)]             */
/*          T not in {T1, .. ,Tn}                                  */
/*  directionnality : in(gr, gr) : out(gr,gr)                      */
/*-----------------------------------------------------------------*/

no_in_renaming(T, []).
no_in_renaming(T, [eq(_, T1) | Renaming]) :-
    T \== T1,
    no_in_renaming(T, Renaming).


/*-----------------------------------------------------------------*/
/*  incl_sub(Left, Right, Substitution)                            */
/*-----------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of terms and/or atoms */
/*          Substitution : list of bindings                        */
/*  relation : there is a substitution Substitution such that      */
/*          Left . Substitution include in Right                   */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)          */
/*-----------------------------------------------------------------*/

incl_sub(Left, Right, Substitution) :-
    incl_sub_1(Left, Right, [], Substitution).


/*-----------------------------------------------------------------*/
/*  incl_sub_1(Left, Right, OldSubstitution, NewSubstitution)      */
/*-----------------------------------------------------------------*/
/*  type :  Left, Right = expressions                              */
/*              = lists of terms and/or atoms                      */
/*          OldSubstitution, NewSubstitution :                     */
/*              lists of bindings                                  */
/*  relation : there is a substitution theta such that             */
/*          Left . θ include in Right                              */
/*          NewSubstitution = [θ | OldSubstitution]                */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)  */
/*-----------------------------------------------------------------*/


incl_sub_1([], Right, OldSubstitution, OldSubstitution).
incl_sub_1([ExprL | Left], [ExprR | Right], OldSubstitution, NewSubstitution) :-
```

```
    compute_substitution_1([ExprL], [ExprR], OldSubstitution,
        NewSubstitution1),
    incl_sub_1(Left, Right, NewSubstitution1, NewSubstitution).
incl_sub_1(Left, [ExprR | Right], OldSubstitution, NewSubstitution) :-
    incl_sub_1(Left, Right, OldSubstitution, NewSubstitution).


/*-----------------------------------------------------------------------*/
/*  incl_ren(Left, Right, Renaming)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of terms and/or atoms      */
/*          Renaming : list of bindings                                  */
/*  relation : there is a renaming Renaming such that                    */
/*          Left . Renaming include in Right                             */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)        */
/*-----------------------------------------------------------------------*/

incl_ren(Left, Right, Renaming) :-
    incl_ren_1(Left, Right, [], Renaming).


/*-----------------------------------------------------------------------*/
/*  incl_ren_1(Left, Right, OldRenaming, NewRenaming)                    */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of terms and/or atoms      */
/*          OldRenaming,  NewRenaming :                                  */
/*              lists of bindings                                        */
/*  relation : there is a renaming Renaming such that                    */
/*          Left . Renaming include in Right                             */
/*          NewRenaming = [Renaming | OldRenaming]                       */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)        */
/*-----------------------------------------------------------------------*/

incl_ren_1([], Right, OldRenaming, OldRenaming).
incl_ren_1([ExprL | Left], [ExprR | Right], OldRenaming, NewRenaming) :-
    compute_substitution_1([ExprL], [ExprR], OldRenaming, NewRenaming1),
    is_renaming(NewRenaming1),
    incl_ren_1(Left, Right, NewRenaming1, NewRenaming).
incl_ren_1(Left, [ExprR | Right], OldRenaming, NewRenaming) :-
    incl_ren_1(Left, Right, OldRenaming, NewRenaming).


/*****************************************************************\
 *  occur_check(Var, Expr)                                         *
 *  type :  Var : variable                                         *
 *          Expr : single expression                               *
 *  relation : Var does not occur in Expr                          *
\*****************************************************************/

occur_check(Var, Expr) :-
    is_var(Expr),
    !,
    Var \== Expr.
occur_check(Var, Expr) :-
        /* not(is_var(Expr)                                         */
    Expr =.. [_ | LT],
    occur_check_list(Var, LT).


/*-----------------------------------------------------------------------*/
/*  occur_check_list(Var, List)                                          */
/*-----------------------------------------------------------------------*/
/*  type :  Var : variable                                               */
/*          List : list of atoms and/or terms                            */
/*  relation : Var does not occur in List                                */
/*  directionnality : in (gr, gr) : out (gr, gr)                         */
/*-----------------------------------------------------------------------*/

occur_check_list(Var, []).
occur_check_list(Var, [T | Ts]) :-
    occur_check(Var, T),
    !,
    occur_check_list(Var, Ts).
```

```
/*-----------------------------------------------------------------------*/
/*  d_unif(Left, Right, Unifier)                                         */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of atoms and/or terms      */
/*          Unifier : list of bindings                                   */
/*  relation : there is a unifier Unifier such that                      */
/*          Left . Unifier = Right . Unifier                             */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)                */
/*-----------------------------------------------------------------------*/

d_unif(Left, Right, Unifier) :-
    d_unif_1(Left, Right, [], Unifier).
```

```
/*-----------------------------------------------------------------------*/
/*  d_unif_1(Left, Right, OldUnifier, NewUnifier)                        */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of atoms and/or terms      */
/*          OldUnifie, NewUnifier : lists of bindings                    */
/*  relation : there is a unifier θ such that                            */
/*          Left . θ = Right . θ                                         */
/*          NewUnifier = [θ | OldUnifier . θ]                            */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)        */
/*-----------------------------------------------------------------------*/

d_unif_1([Left | Lefts], [Left | Rights], OldUnifier, NewUnifier) :-
    is_var(Left),
    !,
    d_unif_1(Lefts, Rights, OldUnifier, NewUnifier).
d_unif_1([Left | Lefts], [Right | Rights], OldUnifier, NewUnifier) :-
        /* not(Left = Right)                                            */
    is_var(Left),
    !,
    occur_check(Left, Right),
    apply_binding(Left, Right, Lefts, NewLefts),
    apply_binding(Left, Right, Rights, NewRights),
    replace_var_sub(Left, Right, OldUnifier, OldUnifier1),
    d_unif_1(NewLefts, NewRights, [eq(Left, Right) | OldUnifier1],
        NewUnifier).
d_unif_1([Left | Lefts], [Right | Rights], OldUnifier, NewUnifier) :-
        /* not(is_var(Left))                                            */
    is_var(Right),
    !,
    occur_check(Right, Left),
    apply_binding(Right, Left, Lefts, NewLefts),
    apply_binding(Right, Left, Rights, NewRights),
    replace_var_sub(Right, Left, OldUnifier, OldUnifier1),
    d_unif_1(NewLefts, NewRights, [eq(Right, Left) | OldUnifier1],
        NewUnifier).
d_unif_1([Left | Lefts], [Right | Rights], OldUnifier, NewUnifier) :-
        /* not(is_var(Left)) and not(is_var(Right))                     */
    Left =.. [F | ListLeft],
    Right =.. [F | ListRight],
    append(ListLeft, Lefts, NewLefts),
    append(ListRight, Rights, NewRights),
    d_unif_1(NewLefts, NewRights, OldUnifier, NewUnifier).
d_unif_1([], [], Unifier, Unifier).
```

```
/*#######################################################*/
/*                    APPLY SUBSTITUTION                 */
/*#######################################################*/


/*-----------------------------------------------------------------------*/
/*  apply_binding(Var, Expr, Old, New)                                   */
/*-----------------------------------------------------------------------*/
/*  type :  Var = variable                                               */
/*          Expr = single expression = term or atom                      */
/*          Old, New = expressions  = lists of terms and/or atoms        */
/*  relation : New is the result of replacing all occurrences            */
/*          of Var (a variable) in Old by Expr                           */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)        */
/*-----------------------------------------------------------------------*/

apply_binding(Var, Expr, [], []).
apply_binding(Var, Expr, [Var | Olds], [Expr | News]) :-
      !,
      apply_binding(Var, Expr, Olds, News).
apply_binding(Var, Expr, [Old | Olds], [Old | News]) :-
            /* not(Old = Var)                                            */
      is_var(Old),
      !,
      apply_binding(Var, Expr, Olds, News).
apply_binding(Var, Expr, [Old | Olds], [New | News]) :-
            /* not(is_var(Old))                                          */
      functor(Old, F, N),
      functor(New, F, N),
      apply_binding_F(N, Var, Expr, Old, New),
      apply_binding(Var, Expr, Olds, News).
```

```
/*-----------------------------------------------------------------------*/
/*  apply_binding_F(N, Var, Expr, Old, New)                              */
/*-----------------------------------------------------------------------*/
/*  type :  N   = integer                                                */
/*          Var  = variable                                              */
/*          Expr = single expression = term or atom                      */
/*          Old  = f(X1, .. ,Xn, .. ,Xmx)                                */
/*          New  = f(Y1, .. ,Yn, .. ,Ymy                                 */
/*  relation : (Y1, .. ,Yn) is the result of remplacing all              */
/*          occurrence of Var (a variable) in (X1, .. ,Xn) by Expr       */
/*  directionnality : in (gr, gr, gr, gr, var) :                         */
/*                out (gr, gr, gr, gr, gr)                               */
/*-----------------------------------------------------------------------*/

apply_binding_F(N, Var, Expr, Old, New) :-
      N>0,
      arg(N, Old, Arg),
      apply_binding(Var, Expr, [Arg], [Arg1]),
      arg(N, New, Arg1),
      N1 is N-1,
      !,
      apply_binding_F(N1, Var, Expr, Old, New).
apply_binding_F(0, Var, Expr, Old, New).


/*-----------------------------------------------------------------------*/
/*  apply_sub_expr(Expr, Substitution, NewExpr)                          */
/*-----------------------------------------------------------------------*/
/*  type :  Expr, NewExpr = lists of terms and/or atoms = expressions    */
/*          Substitution = list of bindings                              */
/*  relation : NewExpr = Expr . Substitution                             */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)                */
/*-----------------------------------------------------------------------*/

apply_sub_expr([], Substitution, []).
apply_sub_expr([Var | Expr], Substitution, [NewVar | NewExpr]) :-
      is_var(Var),
      !,
      ( member_sub(Var, Substitution, NewVar) ->
            apply_sub_expr(Expr, Substitution, NewExpr)
      ;     NewVar = Var,
            apply_sub_expr(Expr, Substitution, NewExpr)
      ).
```

```
apply_sub_expr([E | Expr], Substitution, [NewE | NewExpr]) :-
            /* not(is_var(E))                                          */
      functor(E, F, N),
      functor(NewE, F, N),
      apply_sub_expr_F(N, Substitution, E, NewE),
      apply_sub_expr(Expr, Substitution, NewExpr).


/*-----------------------------------------------------------------------*/
/*  apply_sub_expr_F(N, Substitution, OldFunc, NewFunc)                  */
/*-----------------------------------------------------------------------*/
/*  type :  N : integer                                                  */
/*          Substitution = list of bindings                              */
/*          OldFunc = f(X1, .. ,XN, .. ,Xmx)                             */
/*          NewFunc = f(Y1, .. ,YN, .. ,Ymy)                             */
/*  relation : (Y1, .. ,YN) = (X1, .. ,XN) . Substitution                */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)        */
/*-----------------------------------------------------------------------*/

apply_sub_expr_F(0, Substitution, OldFunc, NewFunc).
apply_sub_expr_F(N, Substitution, OldFunc, NewFunc) :-
      N>0,
      arg(N, OldFunc, Arg),
      apply_sub_expr([Arg], Substitution, [NewArg]),
      arg(N, NewFunc, NewArg),
      N1 is N - 1,
      apply_sub_expr_F(N1, Substitution, OldFunc, NewFunc).
```

```
/*#######################################################################*/
/*                  MANIPULATION OF THE SUBSTITUTIONS                    */
/*#######################################################################*/


/*-----------------------------------------------------------------------*/
/*  replace_var_sub(Var, Term, OldUnification, NewUnification)           */
/*-----------------------------------------------------------------------*/
/*  type :  Var = variable                                               */
/*          Term = term                                                  */
/*          OldUnification, NewUnification : lists of bindings           */
/*  relation : NewUnification is the result of remplacing all occurrence of */
/*              Var (a variable) in Ti of OldUnification by Term         */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)        */
/*-----------------------------------------------------------------------*/

replace_var_sub(Var, Term, [], []).
replace_var_sub(Var, Term, [eq(V, T) | Olds], [eq(V, NewT) | News]) :-
      apply_binding(Var, Term, [T], [NewT]),
      replace_var_sub(Var, Term, Olds, News).


/*-----------------------------------------------------------------------*/
/*  same_substitution(Sub1, Sub2)                                        */
/*-----------------------------------------------------------------------*/
/*  type :  Sub1, Sub2 : lists of bindings                               */
/*  relation : Sub1 = [eq(V1, T1), .. ,eq(Vn, Tn)]                       */
/*          Sub2 = [eq(W1, S1), .. ,eq(Wm, Sm)]                          */
/*          for all Vi = Wj then T1 = Sj                                 */
/*  directionnality : in (gr, gr) : out (gr, gr)                         */
/*-----------------------------------------------------------------------*/

same_substitution([], Sub2).
same_substitution([eq(V, T) | Sub1], Sub2) :-
      member_sub(V, Sub2, T1),
      !,
      T = T1,
      same_substitution(Sub1, Sub2).
same_substitution([S | Sub1],Sub2) :-
            /* not(member_sub(V, Sub2, T1))                              */
      same_substitution(Sub1, Sub2).
```

```
/*-----------------------------------------------------------------------*/
/*  apply_sub_sub(Subst1, Subst2, NewSubst1)                             */
/*-----------------------------------------------------------------------*/
/*  type :  Subst1, Subst2, NewSubst1 : lists of bindings                */
/*  relation : Subst1 = [eq(V1, T1), .. ,eq(Vn, T_n)]                    */
/*             Subst2 = [eq(W1, S1), .. ,eq(Wm, Sm)]                     */
/*             NewSubst1 = [eq(V1, T1'), .. ,eq(Vn, Tn')]                */
/*             Ti' = Ti . Subst2 and it remove it if Ti' = Vi            */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)                */
/*-----------------------------------------------------------------------*/

apply_sub_sub([], Subst2, []).
apply_sub_sub([eq(V, T) | Subst1], Subst2, [eq(V, NewT) | NewSubst1]) :-
      !,
      apply_sub_expr([T], Subst2, [NewT]),
      apply_sub_sub(Subst1, Subst2, NewSubst1).


/*-----------------------------------------------------------------------*/
/*  make_fresh_substitution(ListVar, Substitution, Number, NewNumber)    */
/*-----------------------------------------------------------------------*/
/*  type :  ListVar : list of variables                                  */
/*          Substitution : list of bindings                              */
/*          Number, NewNumber : integers                                 */
/*  relation : ListVar = [V_1, .. ,V_n]                                  */
/*          Substitution = [eq(Vn, Vn.Number), .. ,                      */
/*              eq(V1, V1.Number+n-1)]                                   */
/*          NewNumber = Number + n                                       */
/*  directionnality : in (gr, var, gr, var) : out (gr, gr, gr, gr)       */
/*-----------------------------------------------------------------------*/

make_fresh_substitution([], [], Number, Number).
make_fresh_substitution([Var | ListVar], [eq(Var, NewVar) | Substitution],
            Number, NewNumber) :-
      name(Var, ASCII_Var),
      name(Number, ASCII_Number),
      append(ASCII_Var, ASCII_Number, ASCII_NewVar),
      name(NewVar, ASCII_NewVar),
      Number1 is Number + 1,
      make_fresh_substitution(ListVar, Substitution, Number1, NewNumber).
```

```
/*-----------------------------------------------------------------------*/
/*  member_sub(Var, Substitution, Expr)                                  */
/*-----------------------------------------------------------------------*/
/*  type :  Var : variable                                               */
/*          Substitution : list of bindings                              */
/*          Expr : term or atom                                          */
/*  relation : Substitution = [eq(V1, T1), .. ,eq(Vn, Tn)]               */
/*          there is i such that V = Vi then T = Ti                      */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)                */
/*-----------------------------------------------------------------------*/

member_sub(Var, [eq(Var, Expr) | Substitution], Expr) :-
      !.
member_sub(Var, [S | Substitution], Expr) :-
            /* not(Var = S)                                              */
      member_sub(Var, Substitution, Expr).
```

```
/*#######################################################*/
/*                    GENERAL PREDICATES                 */
/*#######################################################*/


:- compile(library(basics)).
/*    the definitions of member and append                */

/*-----------------------------------------------------------*/
/*    list_var(Expression, OldList, NewList)               */
/*-----------------------------------------------------------*/
/*   type :  Expression = list of terms and/or atoms = expression  */
/*           NewList, OldList = lists of variables         */
/*   relation : NewList is (OldList + the list of Expression's  */
/*                variables) without duplicates            */
/*   directionnality : in (gr, gr, var) : out (gr, gr, gr) */
/*-----------------------------------------------------------*/

list_var([], OldList, OldList).
list_var([Var | Expression], OldList, NewList) :-
      is_var(Var),
      !,
      ( memberchk(Var, OldList) ->
      list_var(Expression, OldList, NewList)
      ;     list_var(Expression, [Var | OldList], NewList)
      ).
list_var([Func | Expression], OldList, NewList) :-
            /* not(is_var(Func))                           */
      Func =.. [_ | NewFunc],
      append(NewFunc,Expression, NewExpression),
      list_var(NewExpression, OldList, NewList).


/*-----------------------------------------------------------*/
/*   is_var(Var)                                           */
/*-----------------------------------------------------------*/
/*   relation :  Var is a variable (Var is of the form $...)  */
/*           36 is the ASCII code of $                     */
/*   directionnality : in (gr) : out (gr)                  */
/*-----------------------------------------------------------*/

is_var(Var) :-
      atomic(Var),
      name(Var, [36 | _]).        /* [36] = '$'            */
:- compile(counter).
```

```
/* compile the counter predicates                           */
/* init_int/1, add_int/2, read_int/2, init_float/2, add_float/2,read_float/2  */


/*#######################################################*/
/* here is the code of the C program that is used to manipulate the counters  */
/*#######################################################*/


#ifndef lint
static char SCCSid[] = "@(#)90/01/14 counter.c ";
#endif

long ctr_int[5];
float ctr_float[5];

void init_integer(Counter)
      int Counter;
{     ctr_int[Counter] = 0;  }

void add_integer(Counter,Value)
      int Counter;
      long Value;
{     ctr_int[Counter] = ctr_int[Counter] + Value;        }

long read_integer(Counter)
      int Counter;
{     return(ctr_int[Counter]);    }

void init_real(Counter)
      int Counter;
{     ctr_float[Counter] = 0;      }


void add_real(Counter,Value)
      int Counter;
      float Value;
{     ctr_float[Counter] = ctr_float[Counter] + Value;   }

float read_real(Counter)
      int Counter;
{     return(ctr_float[Counter]);  }
```

Appendix B

Another meta-interpreter

Here is given the code of the procedure solve and find_new_goal_result presented in subsection 3.3.4. The other procedures of the meta-interpreter are the same as in the previous one.

This meta-interpreter only stops when it finds a solution.

```
/*==================================================================*/
/*  solve(Check, Goal, Resultant, Substitution, ListGoal,           */
/*        ListResult, LastVar, Depth, Derivation)                   */
/*==================================================================*/
/*  type :  Check in { non, empty, evg, eig, evr, eir, svg, sig, svr, sir,  */
/*          (evg,st), (eig,st), (evr,st), (eir,st), (svg,st), (sig,st), (svr,st),  */
/*          (sir,st), (evg,dt), (eig,dt), (evr,dt), (eir,dt), (svg,dt), (sig,dt),  */
/*          (svr,dt), (sir,dt)}.                                     */
/*          Goal, Resultant : expressions                            */
/*          Substitution : list of bindings                          */
/*          ListGoal, ListResult, Derivation : lists of expressions  */
/*          LastVar, Depth : integers                                */
/*  relation :                                                       */
/*  When the derivation is                                           */
/*          G0 => {C1,θ1} ... => Gk-1 => {Ck,θk}Gk                   */
/*  has been constructed, the meanings of the parameters are the following: */
/*          Goal = Gk (also used for loop checking);                 */
/*          Substitution = the list of bindings representing θ1...θk */
/*             restricted to the variables of G0;                    */
/*          LastVar = the number of variables used (needed for       */
/*             standardization apart.)                               */
/*  Input parameters for loop checking:                              */
/*          Check = the loop check that is used;                     */
/*          Resultant = G0S1...Sk;                                   */
/*          ListGoal = [Gk-1, ..., G0];                              */
/*          ListResult = [G0S1...Sk, ..., G0];                       */
/*          Depth = k.                                               */
/*          FI = 1/2 I(I + 1) and 1/2(I - 1)- I < D ≤ 1/2 i (i + 1)  */
/*             if D = FI  then D is a triangular number              */
/*  If a double triangular check is used, ListGoal (ListResult)      */
/*  contains only the goals (resultant heads) with a triangular      */
/*  index.  When "non" is used, these lists are not maintained.      */
/*  ListResult is only maintain for loop checks for resultants       */
/*   or "empty" loop check.                                          */
/*==================================================================*/
```

```
    /* if a solution is found (the goal to be solved is empty)                  */
solve(Check, [], Resultant, Substitution, ListGoal, ListResult,
        LastVar, Depth,I, FI) :-
    nl, write(Substitution).
    /* used to solve built-in predicates, define is syst                        */
solve(Check,  [Literal | Goal], Resultant, Substitution, ListGoal,
        ListResult, LastVar, Depth, I, FI) :-
    syst(Literal), !,
    call_syst(Literal, CAS),
    apply_sub_expr(Goal, CAS, NewGoal),
    apply_sub_expr(Resultant, CAS, NewResultant),
    apply_sub_sub(Substitution, CAS, NewSubstitution),
    update_detph(Depth, I, FI, NewDepth, NewI, NewFI),
    solve(Check, NewGoal, NewResultant, NewSubstitution,
        [NewGoal | ListGoal], [NewResultant | ListResult],
        LastVar, NewDepth, NewI, NewFI).
    /* we perform the loop check (Check) to see if we can continue, then        */
    /* we compute the next goal to be solved and the substitution               */
    /* (NewSubstitution) and the new resultant  (NewResultant) and solve        */
    /* the new goal                                                             */
solve(Check, Goal, Resultant, Substitution, ListGoal, ListResult,
        LastVar, Depth, I, FI) :-
    /* Goal is not empty and ists first literal is not a built-in predicate     */
    check(Check, Goal, Resultant, ListGoal, ListResult,
        Depth, FI, NewListGoal, NewListResult),
    !,
    find_new_goal_resultant(Goal, Resultant, Substitution, LastVar,
        NewGoal, NewResultant, NewSubstitution, NewLastVar),
    update_detph(Depth, I, FI, NewDepth, NewI, NewFI),
    solve(Check, NewGoal, NewResultant, NewSubstitution, NewListGoal,
        NewListResult, NewLastVar, NewDepth, NewI, NewFI).
```

```
/*-----------------------------------------------------------------------------*/
/*  find_new_goal_resultant(PrevGoal, PrevResultant, Substitution,            */
/*          LastVar, NewGoal, NewResultant, NewSubstitution,                  */
/*          NewLastVar)                                                       */
/*-----------------------------------------------------------------------------*/
/*  type :  PrevGoal, PrevResultant, NewGoal, NewResultant : expressions      */
/*          Substitution, NewSubstitution : lists of bindings                 */
/*          LastVar, NewLastVar : integers                                    */
/*  relation : NewGoal is PrevGoal where we replace the first  litteral by its */
/*              body (Body) if it exist else by the atom 'false'              */
/*          NewResultant is PrevResultant to which we apply the               */
/*              unification computes to found Body                            */
/*          The new variables are numered from LastVar to NewLastVar.         */
/*  directionnality : in(gr,  gr, gr, gr, var, var, var, var)                 */
/*                    out(gr, gr, gr, gr, gr, gr, gr, gr)                     */
/*-----------------------------------------------------------------------------*/

find_new_goal_resultant([A | Goal],PrevResultant, Substitution, LastVar,
            NewGoal, NewResultant, NewSubstitution, NewLastVar) :-
        statistics(runtime, _),
    d_clause(A, Goal, NewGoal, SubstitutionClause, LastVar, NewLastVar),
    apply_sub_sub(Substitution, SubstitutionClause, NewSubstitution),
        statistics(runtime, [_ , T]),
        add_float(2, T),
        statistics(runtime, _),
    apply_sub_expr(PrevResultant, SubstitutionClause, NewResultant),
        statistics(runtime, [_ , T1]),
        add_float(1, T1),
        add_int(2, 1).
```

```
/*-----------------------------------------------------------------------------*/
/*  d_clause(Head, Goal, NewGoal, Unifier, Number, NewNumber)                 */
/*-----------------------------------------------------------------------------*/
/*  type :  Head : f(t1, .. ,tn)                                              */
/*              ti = term                                                     */
/*          Goal, NewGoal : expressions                                       */
/*          Unifier : list of bindings                                        */
/*          Number, NewNumber : integers                                      */
/*  relation : if there is a clause in the program (H :- B)                   */
/*              and an unifier Unifier such that                              */
/*              Head . Unifier = H . Unifier                                  */
/*          then Body = B . Unifier                                           */
/*              and all the variables that appear in H and B are              */
/*              renamed, NewNumber - Number is the number of                  */
/*              variables renamed                                             */
/*  directionnality: in (gr, gr, var, var, gr, var) :                         */
/*                   out (gr, gr, gr, gr, gr, gr)                             */
/*-----------------------------------------------------------------------------*/

d_clause(Head, Goal, NewGoal, Unifier, Number, NewNumber) :-
    functor(Head, F, N),
    functor(NewHead, F, N),
    clause(NewHead, B),
    fresh_variables([NewHead | [B]], [H | [B1]], Number, NewNumber),
    d_unif([H], [Head], Unifier),
    append_body_to_goal(B1, Goal, Goal1),
    apply_sub_expr(Goal1,Unifier, NewGoal).
```

Appendix C

The pre-compiler

Here is the code of the pre-compiler presented in the chapter 4. It transforms a PROLOG program into another PROLOG program that uses the loop checking mechanisms (in fact calls the procedure loop_check).

The call to the pre-compiler is

pre_compile(InputFile, OutputFile, UserQuery, Arity)

where

InputFile is the name of the file that contains the program;

OutputFile is the name of the file where the transformed program is written;

UserGoal is the name of the predicate that the user uses as a query qnd Arity its arity.

When the user asks the query he has to add a last parameter which is the kind of loop check to be performed. The procedure loop_check must be loaded with the transformed program.

```
/*##########################################################*/
/*                    Jean HENRARD    14 may 1991           */
/*                    CWI - Amsterdam - The Netherlands     */
/*                    FNDP - Namur - Belgium                */
/*##########################################################*/


/*##########################################################*/
/* There is the structure of the pre_compile program        */
/* pre_compile                                              */
/*    transform_clause                                      */
/*        print                                             */
/*            protray                                       */
/*                write_body                                */
/*        find_dupl                                         */
/*            find_dupl_F                                   */
/*                append                                    */
/*            member_var                                    */
/*        make_new_head                                     */
/*            append                                        */
/*        make_new_body                                     */
/*            make_new_body_1                               */
/*                transform_literal                         */
/*                    sys                                   */
/*                    append                                */
/*            mlb                                           */
/*        add_unif                                          */
/*    make_user_goal                                        */
/*        create_list_var                                   */
/*        append                                            */
/*        print                                             */
/*            protray                                       */
/*                write_body                                */
/*##########################################################*/
```

```
/*==========================================================*\
 *  pre_compile(InputFile, OutputFile, UserGoal, Arity)     *
 *==========================================================*
 *  type :  InputFile, OutputFile, UserGoal : atoms         *
 *          Arity : integer                                 *
 *  relation and side effect                                *
 *          InputFile is the name of an existing file       *
 *          It creates a new clause :                       *
 *              UserGoal(X1, .. ,XArity, Check):-           *
 *                  UserGoal(A1, .. ,XArity, Check,         *
 *                      [[OldHead]], [OldHead],             *
 *                      [OldHead],OldHead,0,ListGoal,       *
 *                      ListResult, LastGoal, Depth).       *
 *          with OldHead = UserGoal(X1, .. ,XArity) and puts it in the file *
 *              OutputFile.                                 *
 *          Transforms the program of file InputFile to a new *
 *          program that uses the loop_checking and puts this new *
 *          program in the file OutputFile after the clause *
 *          "UserGoal (..)"                                 *
 *  directionnality : in (gr, gr, gr, gr) : out (gr, gr, gr, gr) *
\*==========================================================*/
```

```
pre_compile(InputFile, OutputFile, UserGoal, Arity):-
    see(InputFile),
    tell(OutputFile),
    write('/* new program created by transforming the file '),
    write(InputFile),
    write(' */'), nl,
    make_user_goal(UserGoal, Arity),
    read(Clause),
    transform_clause(Clause),
    seen,
    told.
```

```
/*************************************************************\
 *   transform_clause(OldClause)                              *
  ************************************************************
 *   type :  OldClause : a clause                             *
 *   relation : transforms (to use de loop checking) all the clauses  *
 *            from OldClause to the end of the input stream and puts  *
 *            the transform edclauses into the output stream.  *
 *   directionnality : in (gr) : out (gr)                     *
\*************************************************************/

transform_clause('end_of_file'):-
      !.
transform_clause(OldClause):-
            /* OldClause \== 'end_of_file' */
            /* check if OldClause has a body (OldClause is a          */
            /* function with two arguments and ':-' as functor)       */
      name(F, [58, 45]),            /* [58, 45] = ':-'                */
      functor(OldClause, F, 2),
       !,
            /*makes the head of the newclause                         */
      arg(1, OldClause, Head1),
      find_dupl(Head1, [], ListCouple, Head, ListVar),
      make_new_head(Head, Check, ListGoal, ListResult, LastGoal,
            Resultant, Depth,
            LastListGoal, LastListResult, LastLastGoal,
            LastDepth, NewHead),
            /* makes the body of the new clause                       */
      arg(2, OldClause, OldBody),
      make_new_body(OldBody, Check, ListGoal,
            ListResult, LastGoal, Resultant, Depth,
            LastListGoal, LastListResult, LastLastGoal,
            LastDepth, NewBody),
      add_unif(NewBody, ListCouple, NewBody1),
            /* creates the new clause                                 */
      functor(NewClause, F, 2),
      arg(1, NewClause, NewHead),
      arg(2, NewClause, NewBody1),
            /* prints the new clause in the output stream             */
      print(NewClause),
            /* read the next clause in the input stream               */
      read(NextClause),
      transform_clause(NextClause).
```

```
transform_clause(OldClause1):-
            /* OldClause has no body (it isn't a function with        */
            /* ':-' as functor                                        */
            /* makes the head of the newclause                        */
      find_dupl(OldClause1, [], ListCouple, OldClause, ListVar),
      make_new_head(OldClause, Check, ListGoal, ListResult, LastGoal,
            Resultant, Depth,
            LastListGoal, LastListResult, LastLastGoal,
            LastDepth, NewHead),
            /* makes the body of the new clause                       */
      add_unif(loop_check(Check, [], ListGoal, ListResult, LastGoal,
                  Resultant, Depth, LastListGoal, LastListResult,
                  LastLastGoal, LastDepth),
             ListCouple, NewBody),
            /* creates the new clause                                 */
      name(F, [58, 45]),
      functor(NewClause, F, 2),
      arg(1, NewClause, NewHead),
      arg(2, NewClause, NewBody),
            /* prints the new clause in the output stream             */
      print(NewClause),
            /* read the next clause in the input stream               */
      read(NextClause),
      transform_clause(NextClause).
```

```
/*-----------------------------------------------------------------------------*\
 *  make_new_head(Head, Check, ListGoal, ListResult, LastGoal,                 *
 *          Resultant, Depth, LastListGoal, LastListResult,                    *
 *          LastLastGoal, LastDepth, NewHead)                                  *
 *-----------------------------------------------------------------------------*
 *  type :  Head, NewHead : terms                                              *
 *          Check, ListGoal, ListResult, LastGoal, Resultant, Depth,           *
 *          LastListGoal, LastListResult, LastLastGoal, LastDepth :            *
 *              variables                                                      *
 *  relation : Head = f(X1, .. ,Xn)                                            *
 *          NewHead = f(X1, .. ,Xn,Check, ListGoal, ListResult, LastGoal,      *
 *              Resultant, Depth, LastListGoal, LastListResultant,             *
 *              LastLastGoal, LastDepth)                                       *
 *  directionnality :                                                          *
 *          in (novar,var,var,var,var,var,gr,var,var,var,var,var) :             *
 *          out (novar,var,var,var,var,var,gr,var,var,var,gr,novar)             *
\*-----------------------------------------------------------------------------*/

make_new_head(Head, Check, ListGoal, ListResult,
          LastGoal, Resultant, Depth,
          LastListGoal, LastListResult, LastLastGoal, LastDepth,
          NewHead):-
     Head =..ListArg,
     append(ListArg, [Check, ListGoal, ListResult,
          LastGoal, Resultant, Depth,
          LastListGoal, LastListResult,
          LastLastGoal,LastDepth],
          NewListArg),
     NewHead =.. NewListArg.
```

```
/*-----------------------------------------------------------------------------*\
 *  make_new_body(OldBody, Check, ListGoal1, ListResult1, LastGoal1,           *
 *          Resultant, Depth1,                                                 *
 *          LastListGoal, LastListResult, LastLastGoal, LastDepth,             *
 *          NewBody)                                                           *
 *-----------------------------------------------------------------------------*
 * This procedure allows the user to say if he is sure that the clause         *
 * can not generate a loop, he put "no_loop_check" as the first literal        *
 * of the body. Then the precompiler produces a body that does not            *
 * performs the loop check but just updates the differents parameters          *
 * (this is done by using the kind of loop check "non").                       *
 *-----------------------------------------------------------------------------*
 *  type :  OldBody, NewBody :                                                 *
 *          Check, ListGoal1, ListResult1, LastGoal1, Resultant, Depth1,       *
 *          LastListGoal, LastListResult, LastLastGoal, LastDepth :            *
 *              variables                                                      *
 *  relation : OldBody = (b1(Y1), .. ,bn(Yn))                                  *
 *          if b1 = 'no_loop_check'                                            *
 *          then NewBody = (                                                   *
 *              loop_check(non, ListBody, ListGoal, ListResult, LastGoal,      *
 *                  Resultant,Depth,ListGoal1, ListResult1,LastGoal1,          *
 *                  Depth1                                              ),     *
 *              b2(Y2,non,ListGoal1, .. Depth1, ListGoal2, ... ,Depth2),       *
 *              .....                                                          *
 *              bn(Yn,non,ListGoaln, .. ,Depthn,                               *
 *                  LastListGoal, .. ,LastDepth)                        ).     *
 *          else NewBody = (                                                   *
 *              loop_check(Check,ListBody,ListGoal,ListResult,                 *
 *                  LastGoal,Resultant,Depth,ListGoal1,                        *
 *                  ListResult1,LastGoal1,Depth1),                             *
 *              b1(Y1,Check,ListGoal1, .. Depth1,                              *
 *                  ListGoal2, ... ,Depth2                              ),     *
 *              .....                                                          *
 *              bn(Yn,Check,ListGoaln, .. ,Depthn,                             *
 *                  LastListGoal, .. ,LastDepth)                        ).     *
 *  directionnality :                                                          *
 *          in (novar,any,var,var,var,var,gr,var,var,var,var) :                 *
 *          out (novar,any,var,var,var,var,grvar,var,var,novar)                 *
\*-----------------------------------------------------------------------------*/
```

```
make_new_body(('no_loop_check',OldBody), Check, ListGoal, ListResult,
        LastGoal, Resultant, Depth,
        LastListGoal, LastListResult, LastLastGoal, LastDepth,
        NewBody):-
    !,
    make_new_body_1(OldBody, non, ListGoal1, ListResult1, LastGoal1,
        Resultant, Depth1,
        LastListGoal, LastListResult, LastLastGoal, LastDepth,
        NewBody1),
    mlb(OldBody, ListBody),
    NewBody = (loop_check(non, ListBody, ListGoal, ListResult,
            LastGoal, Resultant, Depth, ListGoal1, ListResult1,
            LastGoal1, Depth1),
            NewBody1).
make_new_body('no_loop_check', Check, ListGoal, ListResult, LastGoal,
        Resultant, Depth,
        LastListGoal, LastListResult, LastLastGoal, LastDepth,
        NewBody):-
    !,
    NewBody = loop_check(non, [], ListGoal, ListResult, LastGoal,
            Resultant, Depth,
            LastListGoal, LastListResult, LastLastGoal, LastDepth).
make_new_body(OldBody, Check, ListGoal, ListResult, LastGoal,
        Resultant, Depth,
        LastListGoal, LastListResult, LastLastGoal, LastDepth,
        NewBody):-
         /* OldBody \== (no_loop_check, Body)                    */
    make_new_body_1(OldBody, Check, ListGoal1, ListResult1, LastGoal1,
        Resultant, Depth1,
        LastListGoal, LastListResult, LastLastGoal, LastDepth,
        NewBody1),
    mlb(OldBody, ListBody),
    NewBody = (loop_check(Check, ListBody, ListGoal, ListResult,
            LastGoal, Resultant, Depth, ListGoal1, ListResult1,
            LastGoal1, Depth1),
            NewBody1).
```

```
/*.......................................................................*\
 *  make_new_body_1(OldBody, Check, ListGoal1, ListResult1,            *
 *        LastGoal1, Resultant, Depth1,                                *
 *        LastListGoal, LastListResult, LastLastGoal, LastDepth,       *
 *        NewBody)                                                     *
  *.......................................................................*
 *  type :  OldBody, NewBody :                                         *
 *        Check, ListGoal1, ListResult1, LastGoal1, Resultant,         *
 *        Depth1, LastListGoal, LastListResult, LastLastGoal,          *
 *           LastDepth : variables                                     *
 *  relation : OldBody = (b1(Y1), .. ,bn(Yn))        n >= 1            *
 *        NewBody = (b1(Y1,Check,ListGoal, .. Depth,                   *
 *                ListGoal1, ... ,Depth1                        ),     *
 *           .....                                                     *
 *           bn(Yn,,Check,ListGoaln-1, .. ,Depthn-1,                   *
 *                LastListGoal, .. ,LastDepth)                  ).     *
 *  directionnality :                                                  *
 *        in (novar,any,var,var,var,var,gr,var,var,var,var) :           *
 *        out (novar,any,var,var,var,var,grvar,var,var,novar)           *
\*.......................................................................*/
```

```
make_new_body_1((Arg, Next), Check, ListGoal, ListResult, LastGoal,
        Resultant, Depth,
        LastListGoal, LastListResult, LastLastGoal, LastDepth,
        (NewArg, NewNext)):-
    !,
    transform_literal(Arg, Check, ListGoal, ListResult, LastGoal,
            Resultant, Depth, ListGoal1, ListResult1, LastGoal1,
            Depth1, NewArg),
    make_new_body_1(Next, Check,
        ListGoal1, ListResult1, LastGoal1, Resultant,Depth1,
        LastListGoal, LastListResult, LastLastGoal, LastDepth,
        NewNext).
make_new_body_1(LastArg, Check, ListGoal, ListResult, LastGoal, Resultant,
        Depth,
        LastListGoal, LastListResult, LastLastGoal. LastDepth,
        NewLastArg):-
    transform_literal(LastArg, Check, ListGoal, ListResult, LastGoal,
            Resultant, Depth, LastListGoal, LastListResult,
            LastLastGoal, LastDepth, NewLastArg).
```

```
/*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,*\
 *   transform_literal(Literal, Check, ListGoal, ListResult,                    *
 *              LastGoal, Resultant, Depth, NextListGoal,                       *
 *              NextListResult, NextLastGoal, NextDepth,                        *
 *              newLiteral)                                                     *
  *,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,*
 * This procedure allows to define some "system predicate" that we do          *
 * not have to transform.                                                      *
  *,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,*
 *   type :  Literal, NewLiteral : terms                                       *
 *           Check, ListGoal, ListResult, LastGoal, Resultant, Depth,          *
 *           NextListGoal, NextListResult, NextLastGoal, NextDepth :           *
 *              variables                                                      *
 *   relation : if syst(Literal)                                               *
 *           then  NewLiteral = Literal                                        *
 *              and ListGoal = NextListGoal                                    *
 *              ListResult = NextListResult                                    *
 *              LastGoal = NextLastGoal                                        *
 *              Depth = NextDepth                                              *
 *           else    (not(syst(Literal))), Literal = fX1,  .. , Xn)            *
 *              NewLiteral = f(X1,  .. , Xn, Check, ListGoal, ListResult,      *
 *                    LastGoal, Resultant, Depth, NextListGoal,                *
 *                    NextListResult, NextLastGoal, NextDepth          )       *
 *   directionnality :                                                         *
 *           in(novar,any,var,var,var,var,var,var,var,var,var,var)  :          *
 *           out(novar,any,var,var,var,var,var,var,var,var,var,novar)          *
\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,*/

transform_literal(Literal, Check, ListGoal, ListResult, LastGoal,
           Resultant, Depth, ListGoal, ListResult, LastGoal, Depth,
           Literal):-
      syst(Literal),
      !.
transform_literal(Literal, Check, ListGoal, ListResult, LastGoal,
           Resultant, Depth, NextListGoal, NextListResult, NextLastGoal,
           NextDepth, NewLiteral):-
           /* not(syst(Literal))                                           */
      Literal =.. ListLiteral,
      append(ListLiteral, [Check, ListGoal, ListResult, LastGoal, Resultant,
                  Depth, NextListGoal, NextListResult, NextLastGoal,
                  NextDepth],
           NewListLiteral),
      NewLiteral =.. NewListLiteral.
```

```
syst(unif(L1, L2)).
syst(L1=L2).
syst(!).


/*-------------------------------------------------------------------------------*\
 *   find_dupl(Head, ListVar, ListCouple, NewHead, NewListVar)                  *
 *-----------------------------------------------------------------------------*
 *   type :  Head, NewHead : terms                                              *
 *           ListVar, NewListVar : lists of variables                           *
 *           ListCouple : list of lists of 2 elements                           *
 *   relation : the variables that appear in Head = X1, .. ,Xn                  *
 *           and X11 = X12, .. Xk1 = Xk2                                        *
 *           with        X11, .. ,Xk1 in [X1, .. ,Xn] union ListVar             *
 *              X12, .. ,Xk2 in [X1, .. ,Xn]                                    *
 *           ListCouple = [[X11,X'12], .. ,[Xk1,X'K2]]                          *
 *           NewHead is Head where X12, .. ,Xk2 are replace by                  *
 *              [X'12, .. ,Xk2]                                                 *
 *           X'12, .. ,Xk2 are variables that are unique (new variables)        *
 *   directionnality : in(novar, novar, var, var, var) :                        *
 *                 out(novar, novar, novar, novar, novar)                       *
\*-------------------------------------------------------------------------------*/

find_dupl(Head, ListVar, [[Head, Var]], Var, ListVar):-
      var(Head),
      member_var(Head, ListVar),
      !.
find_dupl(Head, ListVar, [], Head, [Head | ListVar]):-
           /* not(member_var(Head, ListVar))                                 */
      var(Head),
      !.
find_dupl(Head, ListVar, ListCouple, NewHead, NewListVar):-
           /* not(var(Head))                                                 */
      functor(Head, F, N),
      functor(NewHead, F, N),
      find_dupl_F(N, Head, ListVar, ListCouple, NewHead, NewListVar).

find_dupl_F(0, Head, ListVar, [], NewHead, ListVar).
```

```
find_dupl_F(N, Head, ListVar, ListCouple, NewHead, NewListVar):-
      N > 0,
      arg(N, Head, Arg),
      find_dupl(Arg, ListVar, ListCouple1, NewArg, ListVar1),
      arg(N, NewHead, NewArg),
      N1 is N - 1,
      find_dupl_F(N1, Head, ListVar1, ListCouple2, NewHead, NewListVar),
      append(ListCouple1, ListCouple2, ListCouple),
      !.


/*-----------------------------------------------------------------------*\
 *   add_unif(Body, ListCouple, NewBody)                                 *
 *-----------------------------------------------------------------------*
 *   type :  Body, NewBody : clauses                                     *
 *           ListCouple : list of lists of 2 elements                    *
 *   relation : ListCouple = [[X11,X12], .. ,[Xn1,Xn2]]      n>=0        *
 *           Body = (L1, .. ,Lm)      m >=1                              *
 *           NewBodyClause = (unif(X11,X12), .. ,unif(Xn1,Xn2),          *
 *               L1, .. ,Lm)                                             *
 *   directionnality : in(novar, novar, var)                             *
 *                   : out(novar, novar, novar)                          *
\*-----------------------------------------------------------------------*/

add_unif(Body, [], Body).
add_unif(Body, [[X, Y] | ListCouple], (unif(X, Y), NewBody)):-
      add_unif(Body, ListCouple, NewBody).


/*********************************************************************\
 *   make_user_goal(Name, Arity)                                       *
  *********************************************************************
 *   type :  Name : atom                                               *
 *           Arity : integer                                           *
 *   relation : puts on the output stream the clause :                 *
 *           Name(X1, .. ,XArity,Check):-                              *
 *               copy_term([OldHead],OldHeadChange),                   *
 *               Name(X1, .. ,XArity,Check,[OldHeadChange],            *
 *                    OldHeadChange,[OldHead],OldHead,0,               *
 *                    ListGoal,ListResult,LastGoal,Depth).             *
 *           where OldHead = Name(X1, .. ,XArity)                      *
 *   directionnality : in (gr, gr) : out (gr, gr)                      *
\*********************************************************************/
```

```
make_user_goal(Name, Arity):-
          /* creating the head                                          */
      create_list_var(Arity, ListVar),
      append(ListVar, [Check], ArgHead),
      Head =.. [Name | ArgHead],
          /* creating the body                                          */
      OldHead =.. [Name | ListVar],
      append(ListVar, [Check, [OldHeadChange], OldHeadChange, [OldHead],
                OldHead, 0, ListGoal,
                ListResult, LastGoal, Depth],
                ArgNewHead),
      NewHead =.. [Name | ArgNewHead],
          /* creating the clause                                        */
      name(F, [58, 45]),                /* [58, 45] = ':-'              */
      functor(Clause, F, 2),
      arg(1, Clause, Head),
      arg(2, Clause, (copy_term([OldHead], OldHeadChange),
                NewHead)),
      print(Clause).


/*-----------------------------------------------------------------------*\
 *   create_list_var(N, ListVar)                                         *
 *-----------------------------------------------------------------------*
 *   type :  N : integer                                                 *
 *           ListVar : list of variables                                 *
 *   relation : ListVar is a list of N differents variables              *
 *   directionnality : in (gr, var) : out (gr, listvar)                  *
\*-----------------------------------------------------------------------*/

create_list_var(0, []).
create_list_var(N, [Var | ListVar]):-
      N > 0,
      N1 is N -1,
      create_list_var(N1, ListVar).
```

```
/*-----------------------------------------------------------------------*\
 *   portray(Clause)                                                     *
 *-----------------------------------------------------------------------*
 *   type :  Clause : clause                                             *
 *   side effect : prints the clause Clause on the output stream         *
 *           if Clause = h(X) :- b1(Y1),  .. ,bn(Yn)                     *
 *           then it prints                                              *
 *               h(X):-                                                  *
 *                     b1(Y1),                                           *
 *                     ...                                               *
 *                     bn(Yn).                                           *
 *   directionnality : in (gr) : out (gr)                                *
\*-----------------------------------------------------------------------*/

portray(Clause):-
      name(F, [58, 45]),              /* [58, 45] = ':-'                 */
      functor(Clause, F, 2),
      arg(1, Clause, Head),
      write(Head), write(F),
      arg(2, Clause,Body),
      write_body(Body),
      write('.'),
      nl.


/*.......................................................................*\
 *   write_body(Body)                                                    *
  *.......................................................................*
 *   type : Body :                                                       *
 *   side effect : prints Body on the output stream.                     *
 *   directionnality : in (gr) : out (gr)                                *
\*.......................................................................*/

write_body((Arg, Body)):-
      !,
      nl,
      tab(8),
      write(Arg),
      write(','),
      write_body(Body).
write_body(Arg):-
      nl,
      tab(8),
      write(Arg).
```

```
:- compile(library(basics)).
/* the definitions of memberchk and append                              */

member_var(X, [Y | Ls]):-
      X==Y,
      !.
member_var(X, [L | Ls]):-
      member_var(X, Ls).


/*-----------------------------------------------------------------------*/
/*   mlb(Body, ListBody)                                                 */
/*-----------------------------------------------------------------------*/
/*   type :  Body is the body like you recieve it from                   */
/*                clause(Head, Body)                                     */
/*            ListBody : expression                                      */
/*   relation : ListBody = Body transformed in a list                    */
/*-----------------------------------------------------------------------*/

mlb(true, []):-
      !.
mlb(','(B, Body), [B | ListBody]):-
      !,
      mlb(Body, ListBody).
mlb(Body, [Body]).
```

Appendix D

The loop_check procedure

Here is the code of the loop_check procedure that must be used with pre-compiled programs.

```
/*##########################################################*/
/*                    Jean HENRARD    14 may 1991            */
/*                    CWI - Amsterdam - The Netherlands      */
/*                    FNDP - Namur - Belgium                 */
/*##########################################################*/


/*==========================================================*\
 *   loop_check(Check, Body, ListGoal, ListResult, LastGoal,  *
 *              Resultant, Depth, NewListGoal, NewListResult, *
 *              CurrentGoal, NewDepth)                        *
 *==========================================================*
 *   type :  Check in { non, empty, evg, eig, evr, eir, svg, sig, svr, sir,  *
 *           (evg,st), (eig,st), (evr,st), (eir,st), (svg,st), (sig,st), (svr,st),  *
 *           (sir,st), (evg,dt), (eig,dt), (evr,dt), (eir,dt), (svg,dt), (sig,dt),  *
 *           (svr,dt), (sir,dt)}.                                                  *
 *           Body : expression                                                     *
 *           ListGoal, ListResult, NewListGoal, NewListResult :                    *
 *              lists of expressions                                               *
 *           LastGoal, Resultant, CurrentGoal : expressions                        *
 *           Depth, NewDepth : integers                                            *
 *   relation : NewListGoal = the new list of the goals need for the  further      *
 *              loop_check                                                         *
 *           NewListResult = the new list of the resultants need for the           *
 *              further loop_check                                                 *
 *           CurrentGoal is the current goal to be solve = LastGoal with its       *
 *              first element replaces by Body                                     *
 *           NewDepth = Depth + 1                                                  *
 *   directionnality :                                                             *
 *       in  (gr,novar,novar,novar,novar,novar,gr,var,var,var,var)                 *
 *       out  (gr,novar,novar,novar,novar,novar,gr,novar,novar,novar,gr)           *
\*==========================================================*/

loop_check(Check, [], ListGoal, ListResult, [H], Resultant, Depth,
          [[] | ListGoal], ListResult, [], NewDepth):-
      !,
      NewDepth is Depth + 1,
      nl, write('true '),
      nl, write([[] | ListGoal]),
      nl, write(NewDepth).
loop_check(Check, Body, ListGoal, ListResult, [H | Goal], Resultant, Depth,
          NewListGoal, NewListResult, CurrentGoal, NewDepth):-
               /* \+ Body = [] */
      NewDepth is Depth + 1,
      append(Body, Goal, CurrentGoal),
      check(Check, CurrentGoal, Resultant, ListGoal,
            ListResult, NewDepth, NewListGoal, NewListResult),
      !.
loop_check(Check, Body, ListGoal, ListResult, [H | Goal], Resultant, Depth,
            [CurrentGoal | ListGoal], ListResult, CurrentGoal, NewDepth):-
      /* This clause could be removed if we don't want to know when          */
      /* a derivation is pruned                                               */
            /* \+ Body = [] and \+ check( ... )                               */
      NewDepth is Depth + 1/*,
      append(Body, Goal, CurrentGoal),
      nl, write('prune '),
      nl, write([CurrentGoal | ListGoal]),
      nl, write(NewDepth)*/,
      /* the program just said whem he prune but didn't stop                  */
      !, fail.
```

```
/*##################################################*/
/*                THE LOOP CHECKS PROCEDURES                 */
/*##################################################*/

/*==================================================*/
/*  check(Check, Goal, Resultant, ListGoal, ListResult, Depth,      */
/*        NewListGoal, NewListResult)                               */
/*==================================================*/
/*  check use  CASE 1  : /////                                      */
/*             CASE 2  : check_EVG                                  */
/*                       renaming                                   */
/*             CASE 3  : check_EIG                                  */
/*                       compute_substitution                       */
/*             CASE 4  : check_EVR                                  */
/*                       same_EVR                                   */
/*                           renaming                               */
/*                           same_substitution                      */
/*             CASE 5  : check_EIR                                  */
/*                         same_EIR                                 */
/*                           compute_substitution                   */
/*                           same_substitution                      */
/*             CASE 6  : check_SVG                                  */
/*                         incl_ren                                 */
/*             CASE 7  : check_SIG                                  */
/*                         incl_sub                                 */
/*             CASE 8  : check_SVR                                  */
/*                         same_SVR                                 */
/*                           incl_ren                               */
/*                           renaming                               */
/*                           same_substitution                      */
/*             CASE 9  : check_SIR                                  */
/*                         same_SIR                                 */
/*                           incl_sub                               */
/*                           compute_substitution                   */
/*                           same_substitution                      */
/*             CASE 10 : check_t                                    */
/*                         check_EVG, check_EIG, check_EVR,         */
/*                         check_EIR, check_SVG, check_EIG,         */
/*                         check_SVR, check_SIR                     */
/*             CASE 11 : check_t                                    */
/*                         check_EVG, check_EIG, check_EVR,         */
/*                         check_EIR, check_SVG, check_EIG,         */
/*                         check_SVR, check_SIR                     */
/*==================================================*/
```

```
/*==================================================*/
/*  type :  Check in { non, empty, evg, eig, evr, eir, svg, sig, svr, sir,   */
/*          (evg,st), (eig,st), (evr,st), (eir,st), (svg,st), (sig,st), (svr,st),  */
/*          (sir,st), (evg,dt), (eig,dt), (evr,dt), (eir,dt), (svg,dt), (sig,dt),  */
/*          (svr,dt), (sir,dt)}.                                          */
/*          Goal, Resultant : expressions = lists of terms and/or atoms   */
/*          ListGoal, ListResult, NewListGoal, NewListResult :            */
/*                lists of expressions                                    */
/*          depth : integer                                               */
/*  relation : Check is the kind of loop check to be performed            */
/*          Depth is the depth where the loop check occur                 */
/*          if    ListGoal = [G1, .. ,Gn]                                 */
/*                ListResult = [R1, .. ,Rn]                               */
/*          then  there is no i such that Goal 'is sufficiently similar'  */
/*                to Gi w.r.t. Check                                      */
/*                and Resultant 'is sufficiently similar' to Ri w.r.t. Check */
/*                NewListGoal = [Goal | ListGoal]                         */
/*                NewListResult = [resultant | ListResult]                */
/*          exept if Check = ***_d_t then                                 */
/*                if Depth in {1/2 i(i + 1) | i in N}                     */
/*                then NewListGoal = [Goal | ListGoal]                    */
/*                     NewListResult = [Resultant | ListResult]           */
/*                else NewListGoal = ListGoal                             */
/*                     NewListResult = ListResult                         */
/*  directionnality :                                                     */
/*          in(gr, gr, gr, gr, gr, gr, var, var) :                        */
/*          out(gr, gr, gr, gr, gr, gr, gr, gr)                           */
/*==================================================*/

/*****************************************************\
CASE 1                no loop check
\*****************************************************/

check(empty, Goal, Resultant, ListGoal, ListResult, Depth,
          [NewGoal | ListGoal], [NewResultant | ListResult]) :-
     copy_term([Resultant | Goal], [NewResultant | NewGoal]).

check(non, Goal, Resultant, ListGoal, ListResult, Depth, ListGoal,
          ListResult) :-
     copy_term([Resultant | Goal], [NewResultant | NewGoal]).
```

```
/*********************************************************\
CASE 2                Equals Variant of Goal
\*********************************************************/

check(evg, Goal, Resultant, ListGoal, ListResult, Depth,
            ListGoal, ListResult):-
      copy_term([Resultant | Goal], [NewResultant | NewGoal]),
      check_EVG(NewGoal, ListGoal).


/*-----------------------------------------------------------------------*/
/*  check_EVG(Goal, ListGoal)                                            */
/*-----------------------------------------------------------------------*/
/*  type :  Goal : expression                                            */
/*          ListGoal : list of expressions = [G1, .. ,Gn]                */
/*  relation : there is no i and renaming θ  such that Gi θ = Goal       */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

check_EVG(Goal, []).
check_EVG(Goal, [G | ListGoal]):-
      \+ renaming(G, Goal, Renaming),
      check_EVG(Goal, ListGoal).


/*********************************************************\
CASE 3                Equals Instance of Goal
\*********************************************************/

check(eig, Goal, Resultant, ListGoal, ListResult, Depth,
            ListGoal, ListResult):-
      copy_term([Resultant | Goal], [NewResultant | NewGoal]),
      check_EIG(NewGoal, ListGoal).


/*-----------------------------------------------------------------------*/
/*  check_EIG(Goal, ListGoal)                                            */
/*-----------------------------------------------------------------------*/
/*  type :  Goal : expression                                            */
/*          ListGoal : list of expressions = [G_1, .. ,G_n]              */
/*  relation : there is no i and substitution θ  such that Gi θ = Goal   */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

check_EIG(Goal, []).
check_EIG(Goal, [G | ListGoal]):-
      \+ compute_substitution(G, Goal, Substitution),
      check_EIG(Goal, ListGoal).


/*********************************************************\
CASE 4                Equals Variant of Resultant
\*********************************************************/

check(evr, Goal, Resultant, ListGoal, ListResult, Depth,
            [NewGoal | ListGoal], [NewResultant | ListResult]):-
      copy_term([Resultant | Goal], [NewResultant | NewGoal]),
      check_EVR(NewGoal, NewResultant, ListGoal, ListResult).


/*-----------------------------------------------------------------------*/
/*  check_EVR(Goal, Resultant, ListGoal, ListResult)                     */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant : expressions                                */
/*          ListGoal = list of expressions = [G1, .. ,Gn]                */
/*          ListResult = list of expressions = [R1, .. ,Rn]              */
/*  relation : there is no i and renaming θ  such that Gi θ = Goal       */
/*             and Ri θ = Resultant                                      */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

check_EVR(Goal, Resultant, [], []).
check_EVR(Goal, Resultant, [G | ListGoal], [R | ListResult]):-
      \+ same_EVR(Goal, Resultant, G, R),
      check_EVR(Goal, Resultant, ListGoal, ListResult).
```

```
/*-----------------------------------------------------------------------*/
/*  same_EVR(Goal, Resultant, G, R)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant, G, R : expressions                          */
/*  relation : there is a renaming θ  such that G θ = Goal               */
/*             and R θ = Resultant                                       */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

same_EVR(Goal, Resultant, G, R):-
      renaming(G, Goal, RenamingGoal),
      renaming([R], [Resultant], RenamingResultant),
      same_substitution(RenamingResultant, RenamingGoal).


/*********************************************************************\
CASE 5                Equals Instance of Resultant
\*********************************************************************/

check(eir, Goal, Resultant, ListGoal, ListResult, Depth,
            [NewGoal | ListGoal], [NewResultant | ListResult]):-
      copy_term([Resultant | Goal], [NewResultant | NewGoal]),
      check_EIR(NewGoal, NewResultant, ListGoal, ListResult).


/*-----------------------------------------------------------------------*/
/*  check_EIR(Goal, Resultant, ListGoal, ListResult)                     */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant : expressions                                */
/*          ListGoal = list of expressions = [G1, .. ,Gn]                */
/*          ListResult = list of expressions = [R1, .. ,Rn]              */
/*  relation : there is no i and substitution θ  such that Gi θ = Goal   */
/*             and Ri θ = Resultant                                      */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

check_EIR(Goal, Resultant, [], []).
check_EIR(Goal, Resultant, [G | ListGoal], [R | ListResult]):-
      \+ same_EIR(Goal, Resultant, G, R),
      check_EIR(Goal, Resultant, ListGoal, ListResult).
```

```
/*-----------------------------------------------------------------------*/
/*  same_EIR(Goal, Resultant, G, R)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant, G, R : expressions                          */
/*  relation : there is a substitution θ  such that G θ = Goal           */
/*             and R θ = Resultant                                       */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

same_EIR(Goal, Resultant, G, R):-
      compute_substitution(G, Goal, SubstitutionGoal),
      compute_substitution([R], [Resultant], SubstitutionResultant),
      same_substitution(SubstitutionGoal, SubstitutionResultant).


/*********************************************************************\
CASE 6                Subsumes Variant of Goal
/*********************************************************************/

check(svg, Goal, Resultant, ListGoal, ListResult, Depth,
            ListGoal, ListResult):-
      copy_term([Resultant | Goal], [NewResultant | NewGoal]),
      check_SVG(NewGoal, ListGoal).


/*-----------------------------------------------------------------------*/
/*  check_SVG(Goal, ListGoal)                                            */
/*-----------------------------------------------------------------------*/
/*  type :  Goal : expression                                            */
/*          ListGoal : list expressions = [G1, .. ,Gn]                   */
/*  relation : there is no i and renaming θ such that Gi  θ = Goal       */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

check_SVG(Goal, []).
check_SVG(Goal, [G | ListGoal]):-
      \+ incl_ren(G, Goal, Renaming),
      check_SVG(Goal, ListGoal).
```

```
/*************************************************************\
CASE 7                Subsumes Instance of Goal
\*************************************************************/

check(sig, Goal, Resultant, ListGoal, ListResult, Depth,
          ListGoal, ListResult):-
     copy_term([Resultant | Goal], [NewResultant | NewGoal]),
     check_SIG(Goal, ListGoal).


/*-----------------------------------------------------------------------*/
/*  check_SIG(Goal, ListGoal)                                            */
/*-----------------------------------------------------------------------*/
/*  type :  Goal : expression                                            */
/*  ListGoal : list expressions = [G1, .. ,Gn]                           */
/*  relation : there is no i and substitution θ                          */
/*             such that Gi θ = Goal                                     */
/*  directionnality : in(gr, gr) :  out(gr, gr)                          */
/*-----------------------------------------------------------------------*/

/*  check_SIG(Goal, ListGoal)                                            */
/*          true if Goal . Substitution is not include in ListGoal       */

check_SIG(Goal, []).
check_SIG(Goal, [G | ListGoal]):-
     \+ incl_sub(G, Goal, Substitution),
     check_SIG(Goal, ListGoal).


/*************************************************************\
CASE 8                Subsumes Variant of Resultant
\*************************************************************/

check(svr, Goal, Resultant, ListGoal, ListResult, Depth,
          [NewGoal | ListGoal], [NewResultant | ListResult]):-
     copy_term([Resultant | Goal], [NewResultant | NewGoal]),
     check_SVR(NewGoal, NewResultant, ListGoal, ListResult).
```

```
/*-----------------------------------------------------------------------*/
/*  check_SVR(Goal, Resultant, ListGoal, ListResult)                     */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant : expressions                                */
/*          ListGoal = list of expressions = [G1, .. ,Gn]                */
/*          ListResutl = list of expressions = [R1, .. ,Rn]              */
/*  relation : there is no i and renaming θ such that Gi θ include in Goal */
/*             and Ri θ = Resultant                                      */
/*  directionnality : in (gr, gr, gr, gr) : out (gr, gr, gr, gr)         */
/*-----------------------------------------------------------------------*/

check_SVR(Goal, Resultant, [], []).
check_SVR(Goal, Resultant, [G | ListGoal], [R | ListResult]):-
     \+ same_SVR(Goal, Resultant, G, R),
     check_SVR(Goal, Resultant, ListGoal, ListResult).


/*-----------------------------------------------------------------------*/
/*  same_SVR(Goal, Resultant, G, R)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant, G, R : expressions                          */
/*  relation : there is a renaming θ such that G θ include in Goal       */
/*             and R θ = Resultant                                       */
/*  directionnality : in (gr, gr, gr, gr) : out (gr, gr, gr, gr)         */
/*-----------------------------------------------------------------------*/

same_SVR(Goal, Resultant, G, R):-
     incl_ren(G, Goal, RenamingGoal),
     renaming([R], [Resultant], RenamingResultant),
     same_substitution(RenamingResultant, RenamingGoal).


/*************************************************************\
CASE 9                Subsumes Instance of Resultant
\*************************************************************/

check(sir, Goal, Resultant, ListGoal, ListResult, Depth,
          [NewGoal | ListGoal], [NewResultant | ListResult]):-
     copy_term([Resultant | Goal], [NewResultant | NewGoal]),
     check_SIR(NewGoal, NewResultant, ListGoal, ListResult),
     !.
```

```
/*-----------------------------------------------------------------------*/
/*  check_SIR(Goal, Resultant, ListGoal, ListResult)                     */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant : expressions                                */
/*          ListGoal = list of expressions = [G1, .. ,Gn]                */
/*          ListResult = list of expressions = [R1, .. ,Rn]              */
/*  relation : there is no i and substitution θ such that Gi θ include in Goal  */
/*             and Ri θ = Resultant                                      */
/*  directionnality : in (gr, gr, gr, gr) : out (gr, gr, gr, gr)         */
/*-----------------------------------------------------------------------*/

check_SIR(Goal, Resultant, [], []).
check_SIR(Goal, Resultant, [G | ListGoal], [R | ListResult]):-
      \+ same_SIR(Goal, Resultant, G, R),
      check_SIR(Goal, Resultant, ListGoal, ListResult).


/*-----------------------------------------------------------------------*/
/*  same_SIR(Goal, Resultant, G, R)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Goal, Resultant, G, R : expressions                          */
/*  relation : there is a substitution θ such that G θ include in Goal   */
/*             and R θ = Resultant                                       */
/*  directionnality : in (gr, gr, gr, gr) : out (gr, gr, gr, gr)         */
/*-----------------------------------------------------------------------*/

same_SIR(Goal, Resultant, G, R):-
      incl_sub(G, Goal, SubstitutionGoal),
      compute_substitution([R], [Resultant], SubstitutionResultant),
      same_substitution(SubstitutionResultant, SubstitutionGoal).
```

```
/*********************************************************\
CASE 10              Single triangle loop checks
\*********************************************************/

:- compile(library(math)).

check((Full,st), Goal, Resultant, ListGoal, ListResult, Depth,
          [NewGoal | ListGoal], [NewResultant | ListResult]):-
     copy_term([Resultant | Goal], [NewResultant | NewGoal]),
     Value is (1+(8 * Depth)),
     sqrt(Value, I),
     ( I is float(integer(I)) ->
          check_t(Full, NewGoal, [NewResultant], ListGoal, ListResult)
     ;    true
     ).


/*********************************************************\
CASE 11              double triangle loop checks
\*********************************************************/

check((Full,dt), Goal, Resultant, ListGoal, ListResult, Depth,
          NewListGoal,NewListResult) :-
     Value is (1+(8 * Depth)),
     sqrt(Value, I),
     ( I is float(integer(I)) ->
          copy_term([Resultant | Goal], [NewResultant | NewGoal]),
          check_t(Full, NewGoal, [NewResultant], ListGoal, ListResult),
          NewListGoal =  [NewGoal | ListGoal],
          NewListResult = [NewResultant | ListResult]
     ;    NewListGoal = ListGoal,
     NewListResult = ListResult
     ).
```

```
/*-----------------------------------------------------------------------------*/
/*  check_t(Check, Goal, Resultant, ListGoal, ListResult)                      */
/*-----------------------------------------------------------------------------*/
/*  type :  Check in {non, evg, eig,evr,eir,svg,sig,svr,sir}                   */
/*          Goal,  Resultant : expressions                                     */
/*          ListGoal, ListResult : lists of expressions                        */
/*  relation : ListGoal = [G1, .. ,Gn]                                         */
/*          ListResult = [R1, .. ,Rn]                                          */
/*          there is no i such that Goal, Resultant is sufficiently            */
/*              similar to Gi,Ri w.r.t. Check                                  */
/*  directionnality : in (gr, novar, novar, novar, novar)                      */
/*              : out (gr, novar, novar, novar, novar)                         */
/*-----------------------------------------------------------------------------*/

check_t(evg, Goal, Resultant, ListGoal, ListResult):-
      check_EVG(Goal, ListGoal).
check_t(eig, Goal, Resultant, ListGoal, ListResult):-
      check_EIG(Goal, ListGoal).
check_t(evr, Goal, Resultant, ListGoal, ListResult):-
      check_EVR(Goal, Resultant, ListGoal, ListResult).
check_t(eir, Goal, Resultant, ListGoal, ListResult):-
      check_EIR(Goal, Resultant, ListGoal, ListResult).
check_t(svg, Goal, Resultant, ListGoal, ListResult):-
      check_SVG(Goal, ListGoal).
check_t(sig, Goal, Resultant, ListGoal, ListResult):-
      check_SIG(Goal, ListGoal).
check_t(svr, Goal, Resultant, ListGoal, ListResult):-
      check_SVR(Goal, Resultant, ListGoal, ListResult).
check_t(sir, Goal, Resultant, ListGoal, ListResult):-
      check_SIR(Goal, Resultant, ListGoal, ListResult).
```

```
/*#############################################################*/
/*    COMPUTE SUBSTITUTION, RENAMING, UNIFICATION,             */
/*#############################################################*/

/*-----------------------------------------------------------------------------*/
/*  compute_substitution(Left, Right, Substitution)                            */
/*-----------------------------------------------------------------------------*/
/*  type :  Left, Right = expressions                                          */
/*              = lists of terms and/or atoms                                  */
/*          Substitution : list of bindings                                    */
/*  relation : there is a substitution Substitution such that                  */
/*              Left . Substitution = Right                                    */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)                      */
/*-----------------------------------------------------------------------------*/

compute_substitution(Left, Right, Substitution):-
      compute_substitution_1(Left, Right, [], Substitution).


/*-----------------------------------------------------------------------------*/
/*  compute_substitution_1(Left, Right, FirstSubstitution,                     */
/*          Substitution)                                                      */
/*-----------------------------------------------------------------------------*/
/*  type :  Left, Right = expressions                                          */
/*              = lists of terms and/or atoms                                  */
/*          FirstSubstitution, Substitution : list of bindings                 */
/*  relation : there is a substitution θ such that Left . θ = Right            */
/*          Substitution = [θ | FirstSubstitution]                             */
/*              without duplicates                                             */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)              */
/*-----------------------------------------------------------------------------*/

compute_substitution_1([], [], FirstSubstitution, FirstSubstitution):-
      !.
compute_substitution_1([L | Left], [R | Right], FirstSubstitution,
          Substitution) :-
      var(L),
      !,
      ( member_sub(L, FirstSubstitution, Term) ->
          R == Term,
          compute_substitution_1(Left, Right, FirstSubstitution, Substitution)
      ;   compute_substitution_1(Left, Right, [eq(L, R) | FirstSubstitution],
              Substitution)
      ).
compute_substitution_1([L | Left], [R | Right], FirstSubstitution,
```

```
        Substitution):-
        /* \+ var(L)                                    */
    functor(L, F, N),
    \+ var(R),
    functor(R, F, N),
    L =.. [F | ListL],
    append(ListL, Left, NewLeft),
    R =.. [F | ListR],
    append(ListR, Right, NewRight),
    compute_substitution_1(NewLeft, NewRight, FirstSubstitution,
        Substitution).


/*-----------------------------------------------------------------------*/
/*  renaming(Left, Right, Renaming)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of terms and/or atoms      */
/*          Renaming : list of bindings                                  */
/*  relation : there is a renaming Renaming such that                    */
/*             Left . Renaming = Right                                   */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)                */
/*-----------------------------------------------------------------------*/

renaming(Left, Right, Renaming):-
    compute_substitution(Left, Right, Renaming),
    compute_substitution(Right, Left, Renaming1).


/*-----------------------------------------------------------------------*/
/*  incl_sub(Left, Right, Substitution)                                  */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions                                    */
/*                = lists of terms and/or atoms                          */
/*          Substitution : list of binding substitutions                 */
/*  relation : there is a substitution Substitution such that            */
/*          Left . Substitution include in Right                         */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)                */
/*-----------------------------------------------------------------------*/

incl_sub(Left, Right, Substitution):-
    incl_sub_1(Left, Right, [], Substitution).


/*-----------------------------------------------------------------------*/
/*  incl_sub_1(Left, Right, OldSubstitution, NewSubstitution)            */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of terms and/or atoms      */
/*          OldSubstitution, NewSubstitution :                           */
/*              lists of bindings                                        */
/*  relation : there is a substitution theta such that                   */
/*          Left . θ include in Right                                    */
/*          NewSubstitution = [θ | OldSubstitution]                      */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)        */
/*-----------------------------------------------------------------------*/

incl_sub_1([], Right, OldSubstitution, OldSubstitution).
incl_sub_1([ExprL | Left], [ExprR | Right], OldSubstitution, NewSubstitution):-
    compute_substitution_1([ExprL], [ExprR], OldSubstitution,
        NewSubstitution1),
    incl_sub_1(Left, Right, NewSubstitution1, NewSubstitution).
incl_sub_1(Left, [ExprR | Right], OldSubstitution, NewSubstitution):-
    incl_sub_1(Left, Right, OldSubstitution, NewSubstitution).


/*-----------------------------------------------------------------------*/
/*  incl_ren(Left, Right, Renaming)                                      */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions  = lists of terms and/or atoms     */
/*          Renaming : list of bindings                                  */
/*  relation : there is a renaming Renaming such that                    */
/*          Left . Renaming include in Right                             */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)        */
/*-----------------------------------------------------------------------*/

incl_ren(Left, Right, Renaming):-
    incl_ren_1(Left, Right, [], Renaming).
```

```
/*-----------------------------------------------------------------------*/
/*  incl_ren_1(Left, Right, OldRenaming, NewRenaming)                    */
/*-----------------------------------------------------------------------*/
/*  type :  Left, Right = expressions = lists of terms and/or atoms       */
/*          OldRenaming,  NewRenaming : lists of bindings                 */
/*  relation : there is a renaming Renaming such that                     */
/*          Left . Renaming include in Right                              */
/*          NewRenaming = [Renaming | OldRenaming]                        */
/*  directionnality : in (gr, gr, gr, var) : out (gr, gr, gr, gr)         */
/*-----------------------------------------------------------------------*/

incl_ren_1([], Right, OldRenaming, OldRenaming).
incl_ren_1([ExprL | Left], [ExprR | Right], OldRenaming, NewRenaming):-
      renaming([ExprL], [ExprR], Renaming),
      same_substitution(Renaming, OldRenaming),
      append(Renaming, OldRenaming, OldRenaming1),
      incl_ren_1(Left, Right, OldRenaming1, NewRenaming).
incl_ren_1(Left, [ExprR | Right], OldRenaming, NewRenaming):-
      incl_ren_1(Left, Right, OldRenaming, NewRenaming).


/*=======================================================================*\
 *  unif(Left, Right)                                                    *
/*=======================================================================*\
 *  type :  Left, Right : expressions                                    *
 *  relation : Left = Right (+ occur check)                              *
\*=======================================================================*/

unif(Left, Right):-
      unif_1([Left], [Right]).
unif_1([L | Left], [R | Right]):-
      var(L),
      var(R),
      !,
      R = L,
      unif_1(Left, Right).
unif_1([L | Left], [R | Right]):-
            /* \+ var(R)                                                 */
      var(L),
      !,
      occur_check(L, R),
      L = R,
      unif_1(Left, Right).
unif_1([L | Left], [R | Right]):-
            /* \+ var(L)                                                 */
      var(R),
      !,
      occur_check(R, L),
      R = L,
      unif_1(Left, Right).
unif_1([L | Left], [R | Right]):-
            /* \+ var(L) and \+ var(R)                                   */
      functor(L, F, NL),
      L =.. [F | ListL],
      functor(R, F, NL),
      R =.. [F | ListR],
      append(ListL, Left, NewLeft),
      append(ListR, Right, NewRight),
      unif_1(NewLeft, NewRight).
unif_1([], []).


/*********************************************************************\
 *  occur_check(Var, Expr)                                           *
  *******************************************************************
 *  type :  Var : variable                                           *
 *          Expr : single expression                                 *
 *  relation : Var does not occur in Expr                            *
\*********************************************************************/

occur_check(Var, Expr):-
      var(Expr),
      !,
      Var \== Expr.
occur_check(Var, Expr):-
      Expr =.. [F | LT],
      occur_check_list(Var, LT).


occur_check_list(Var, []).
occur_check_list(Var, [T | Ts]):-
      occur_check(Var, T),
      occur_check_list(Var, Ts).
```

```
/*#######################################################*/
/*      MANIPULATION OF THE SUBSTITUTIONS                */
/*#######################################################*/


/*-------------------------------------------------------------------------------*/
/*  same_substitution(Sub1, Sub2)                                                */
/*-------------------------------------------------------------------------------*/
/*  type :  Sub1, Sub2 : lists of bindings                                       */
/*  relation : Sub1 = [eq(V1, T1), .. ,eq(Vn, Tn)]                               */
/*             Sub2 = [eq(W1, S1), .. ,eq(Wm, Sm)]                               */
/*             for all Vi = Wj then T1 = Sj                                      */
/*  directionnality : in (gr, gr) : out (gr, gr)                                 */
/*-------------------------------------------------------------------------------*/

same_substitution([], Sub2).
same_substitution([eq(V, T) | Sub1], Sub2):-
      member_sub(V, Sub2, T1),
      !,
      T == T1,
      same_substitution(Sub1, Sub2).
same_substitution([S | Sub1], Sub2):-
      same_substitution(Sub1, Sub2).


/*-------------------------------------------------------------------------------*/
/*  member_sub(Var, Substitution, Expr)                                          */
/*-------------------------------------------------------------------------------*/
/*  type :  Var : variable                                                       */
/*          Substitution : list of bindings                                      */
/*          Expr : term or atom                                                  */
/*  relation : Substitution = [eq(V1, T1),  .. , eq(Vn, Tn)]                     */
/*          there is i such that V = Vi then T = Ti                              */
/*  directionnality : in (gr, gr, var) : out (gr, gr, gr)                        */
/*-------------------------------------------------------------------------------*/

member_sub(Var, [eq(V, Expr) | Substitution], Expr):-
      Var == V,
      !.
member_sub(Var,[S | Substitution], Expr):-
            /* \+ Var == S                                                       */
      member_sub(Var, Substitution, Expr).
```

```
/*#######################################################*/
/*                GENERAL PREDICATES                     */
/*#######################################################*/

:- compile(library(basics)).
```

Bibliography

[A 88] K.R. Apt, *Introduction to Logic Programming (revised and extended version)*, Technical Report CS-R8826, Centre for Mathematics and Computer Sciences, Amsterdam, 1988.

[BAK 89] R.N. Bol, K.R. Apt, J.W. Klop, *An Analysis of Loop Checking Mechanisms for Logic Programs*, Technical Report CS-R8942, Centre for Mathematics and Computer Sciences, Amsterdam, 1989.

[B 90] R.N. Bol, *Towards more efficient loop checks*, in: Proceeding of the 1990 North American conference on Logic Programming (S. Debray, M. Hermenegildo eds.), MIT Press, Cambridge Massachussets, 1990, 465-479.

[D 90] Y. Deville, *Logic Programming, systematic program development*, Addison Wesley, 1990.

[L 87] J.W. Lloyd, *Foundations of Logic Programming*, Second Edition, Springer-Verlag, Berlin, 1987.

[MM 82] A. Martelli, U. Montanari, *Am Efficient Unification Algorithm*, ACM transactions on Programing languages and systems, Vol 4, No 2, 1982, 258-282.

[NS 91] J.F. Naughton, Y. Sagiv, *A simple characterization of uniform boundness for class of recursions*, journal of logic programming, Vol 10, No 3 & 4, 1991, 233-253.

[P 87] *C-PROLOG User's Manual*, Version 1.5, November 26, 1987

[R 65] J.A. Robinson, *A machine-oriented logic based on the resolution principle*, JACM, 12:1, January 1965, 23-41.

[SS 86] L. Sterling and E. Shapiro, *The Art of PROLOG : Advanced Programming Techniques*, Cambridge, MA : MIT PRESS

[S 88] L. Sterling, *Constructing meta-interpreter for logic programs*, Lecture notes, Advanced School on Fondations of Logic Programming, Alghero, Sardinia, Italy, September 1988.

[S 89] M.E. Stickel, *A PROLOG Technology Theorem Prover : A New Exposition and Implementation in PROLOG*, Technical note 464, Artificial Intelligence Center, Computer and Information Sciences Division, SRI International, June 89.

[W 84] D.H.D. Warren, *An abstract PROLOG instruction set*, Report 309, Artificial Intelligence Center, SRI international, Menlo Park, CA, October 1984.