Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

A new procedural semantics for prolog

Trinon, Jean-Marc

Award date:
1987

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 25. Apr. 2024

A NEW PROCEDURAL SEMANTICS FOR PROLOG

-

Jean-Marc TRINON

Mémoire présenté en vue de l'obtention du grade de Licencié et Maître en Informatique.

Promoteur : Baudouin LE CHARLIER

Année Académique 1986-1987

ACKNOWLEDGEMENTS

I am particularly grateful to Mr. Baudouin Le Charlier, supervisor of this work, for tens of hours of questions, discussions and arguments about the material of this work. His guidance and his advice have greatly improved the quality of the results. I take this opportunity to offer my deepest thanks.

I would also like to thank Mr. Yves Deville for his useful comments on earlier drafts.

I take great pleasure in acknowledging Mr. Michel Vanden Bossche-Marquette and all the members of B.I.M. company in Everberg. The time I spent there has greatly enhanced my knowledge of PROLOG. Thanks also for allowing me to use their text processing tools.

I owe a great debt of thanks to my parents for giving me full support during my university education.

Finally, I wish to send my apologies to all the people I have disturbed during the elaboration of this work.

CONTENTS

PROPOSITIONS AND LEMMAS

FIGURES

INTRODUCTION

Nowadays, some PROLOG-systems are becoming actual development systems. The time has come when the usage of this language is no longer restricted to a small number of AI "laboratories" . The need for a methodology of development for PROLOG programs is therefore urgent. Ideally, this methodology should be as close as possible to the intuitive approach of PROLOG programmers.

As PROLOG is a consequence activities in the field of "Logic Programming", it is interesting to quickly browse through the different methodologies developed in this field.

One trend tries to develop the knowledge and the theories concerning predicate logic. Kowalski's work [Kowalski 79] is a good illustration of this trend. But we shall see in chapter 1 that programming in PROLOG is different from programming in logic !

A second trend is illustrated by Deville's work [Deville 87]. This trend consists of separating the logical aspects from the non-logical ones. It tries to reconciliate declarative and procedural semantics. It suggests that programming in PROLOG is programming in logic but augmented with something else (control information for example). So, in the methodology, a first step is only concerned with logical aspects and only during the second one, non-logical aspects are taken into account.

Finally, a third trend gives priority to the procedural aspects of the logic programming language. It believes that the gap between the declarative and the procedural semantics is so huge that it is simpler to build correct programs by using almost exclusively procedural aspects. But the usual procedural semantics founded on the search for solutions in a tree (as explained in [Lloyd 84]) is not usable for the development of a practical methodology.

A first methodology based on this third trend has been developed last year at the University of Namur [Derroitte 86]. In that work, the procedural semantics of a subset of PROLOG (a purely deterministic subset without the cut) is discussed and a methodology for proof of correctness is developed. The determinism simplifies the problem because no *backtracking* can occur .

Our work comes as an extension of [Derroitte 86] in the sense that we suggest a procedural semantics for the whole PROLOG language even in its non-deterministic

aspects. The procedural semantics we propose is founded on three algorithms presented in chapter 2 ; it is as close as possible to the intuitive approach of PROLOG programmers.

In chapter 1 , we recall some fundamental concepts of logic programming. Chapters 3 and 4 establish the equivalence between our semantics and the usual one for finite and infinite (respectively) SLD-trees without cuts. Chapter 5 investigates the problem of cuts.

Chapter 6 is concerned with possible future researchs towards the elaboration of a frame of specification, the introduction of extra-logical features and the problem of proofs of correctness. These aspects must be studied in order to conceive a complete methodology.

CHAPTER 1 : LOGIC PROGRAMMING

1.1 INTRODUCTION :

The "key idea" of logic programming is that logic can be used as a programming language. This was introduced in the early 70's mainly by Kowalski [Kowalski 74] and Colmerauer, although Green should also be mentioned [Green 69]. It comes as consequence of earlier works in the field of automatic theorem-proving and particularly of Robinson's landmark paper [Robinson 65] about the resolution principle, resolution being an inference rule well-suited to automation on computer.

Up to now, the PROLOG language has been the major outgrowth of the logic programming paradigm. PROLOG is the acronym of PROgramming in LOGic.

In this chapter, we begin with a review of the syntax of logic programs based on the syntax of Horn clauses. We define there some notational and denominational conventions.

Then, we turn to the definition of the declarative semantics of logic programs. This semantics provides the interpretation of programs at the logical level.

Next, we examine the procedural semantics which deals with the procedural interpretation of Horn clause logic. This interpretation makes Horn clause logic very effective as a programming language. We also recall the fundamental theorems establishing the equivalence of the two semantics.

Finally, we show how this procedural semantics can be used for the PROLOG programs but also how some PROLOG features can destroy the equivalence of the two semantics.

Much of this chapter is inspired from earlier works : [Deville 87], [Lloyd 84] and [Kowalski 79].

1.2 SYNTAX :

This aspect is concerned with the syntactic definition of a first order language. We recall only some key definitions.

1.2.1 Alphabet :

First, an *alphabet* must be defined. It will be used to build the well-formed formulas of the language. The alphabet can be subdivised into seven classes of symbols :

- constants
- variables
- functions
- predicates
- connectives
- quantifiers
- punctuation symbols

Usually, the first four classes vary from alphabet to alphabet, while the others remain the same.

The set of *constants* is composed of finite strings of characters. The set of *variables* is composed of finite strings of letters and digits preceeded by an underscore (convention). The set of variables and constants must be disjoint.

The sets of n-ary *functions* and n-ary *predicates* are composed of finite strings of letters, digits and special characters (the set of special characters being {+,-,*,/,=,<,>, •}) . Each string will be subscripted with <function,n> in the n-ary function set and with <predicate,n> in the n-ary predicate set.

The set of *connectives* is { $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ }.

The set of *quantifiers* is { $\exists$, $\forall$ }.

The set of *punctuation symbols* is { (,) , ,}.

We use some notational conventions :

- for readability, we allow the insertion of the underscore (_) anywhere in constants and in the middle of variables and functions.
- the sets of n-ary predicates and n-ary functions are all disjoint (because of their subscripts). When writing formulas, we drop these subscripts. The distinction between functions and predicates is clear in every formula. Moreover, the arity is also unambiguous. But is the same string stands for a predicate and a function (or two predicates or functions with different arity), they are conceptually different because of their virtual subscripts.

1.2.2 First order language :

We can now define the *first order language* given by an alphabet: it consists of the set of all well-formed formulas constructed from the symbols of the alphabet.

The syntax of *well-formed formulas* (wff) complies to the following rules :

<wff> ::= <atomic formula> |
¬ <wwf> |
<wwf> ∧ <wwf> |
<wwf> ∨ <wwf> |
<wwf> → <wwf> |
<wwf> ↔ <wwf>
∃ <variable> <wwf> |
∀ <variable> <wwf>

<atomic formula> ::= <predicate> | <predicate> (<list of terms>)

<list of terms> ::= <term> | <term> , <list of terms>

<term> ::= <constant> | <variable> | <function> |
<function> (<list of terms>)

Later on, when we speak of a formula, it means a well-formed formula.

Examples :

Assume that
a, b are constants,
f, g are functions,
p, q are predicates,

_x, _y are variables.

The following are formulas :

$\forall$_x p(_x)

$\forall$_x ($\exists$_y (p(_x , f(_y)) $\rightarrow$ q(_x)))

$\neg$ ($\exists$_x (p(_x , a) $\wedge$ q(f(a))))

Notational conventions :

- often, F $\rightarrow$ G will be written as G $\leftarrow$ F .
- we also write the term •(_h , _t) as [_h | _t], and [t1,t2, ... ,tn] for •(t1, •(t2, (..., •(tn,[]) ...))) when t1, t2, ... , tn are terms.
- to avoid bracketting as much as possible, we adopt the following order of precedence (highest at top) with a left to right associativity rule :

  $\neg$, $\exists$, $\forall$

  $\wedge$

  $\vee$

  $\rightarrow$, $\leftrightarrow$

- for some predicates it is also handfull to use an infix notation instead of a prefix notation. For instance we write t1=t2 in place of =(t1,t2) where t1 and t2 are terms.

In $\forall$_x p(_x) or $\exists$_x p(_x) , the scope of $\forall$_x or $\exists$_x, respectively, is p(_x) . An occurrence of a variable immediately following a quantifier or within the scope of a quantifier concerning this variable is a bound occurrence. Any other occurrence of a variable is free.

We define a *closed formula* as a formula with no free occurrence of any variable.

If L is an atomic formula, L and $\neg$ L are *literals*. L is a positive literal. $\neg$ L is a negative literal.

1.2.3 Clause :

Then, we define a *clause* as a formula of the form

$$\forall\_x_1 \ldots \forall\_x_s \ (L_1 \vee \ldots \vee L_m)$$

where each L_i is a literal and $\_x_1$, ... , $\_x_s$ are the variables occurring in ($L_1 \vee \ldots \vee L_m$)

A *Horn clause* is a clause with one or no positive literal.

We adopt the usual shorthand notation for the clauses, so the clause

$$\forall \_x_1 \ldots \forall \_x_s (A_1 \vee \ldots \vee A_k \vee \neg B_1 \vee \ldots \vee \neg B_m)$$

where A_1, ... , A_k , B_1 , ... , B_m are atomic formulas and $\_x_1$, ... ,$\_x_s$ are all the variables occurring in these atoms, is denoted by

$$A_1, \ldots , A_k \leftarrow B_1, \ldots , B_m$$

A_1, ... , A_k is called the *consequent* and B_1, ... ,B_m the *antecedent* .

1.2.4 Logic program, program clause, goal clause :

Now, we can turn to the definition of a *logic program* : it is a finite set of program clauses, a program clause being a Horn clause with one positive literal.

So, a *program clause* has the following form :

$$A \leftarrow B_1 , \ldots , B_n$$

A is called the *head* (or consequent) and B_1 , ... , B_n the *body* (or antecedent) of the program clause. A program clause with an empty body is also called a *unit clause*.

The set of all program clauses, of a logic program, with the same predicate in the head is called the definition of this predicate.

A clause with an empty consequent is called a *goal clause*. It has the form

$$\leftarrow B_1 , \ldots , B_n$$

and each B_i (i=1 ... n) is a *subgoal* of the goal clause.

Note : we do not call a goal clause simply a goal ! When we speak of a goal, we mean the body of a goal clause, thus $(B_1 , ... , B_n)$. A goal is then a list of subgoals.

The empty clause (the clause with empty consequent and empty antecedent) is denoted $\Diamond$.

1.3 DECLARATIVE SEMANTICS :

In this section, we briefly discuss the meaning which can be attached to a logic program. We do not pay too much attention to this aspect because our work is mostly concerned with the procedural interpretation of logic programs. However, some key concepts are important.

The declarative semantics provides an understanding of the program in terms of formulas and truth values in first order logic. We do not agree with the assertion saying that the declarative semantics of a program provides its specification because we think a (easily understandable) specification requires some concepts which can not be embraced in a simple way by first order logic. Possibly, the declarative semantics can be seen as the translation of the specification, or of some of its aspects, under the form of logic formulas. For more details about this question, we suggest to consult [Le Charlier 85].

1.3.1 Intuitive presentation :

The quantifiers and connectives have the following meanings :

$\neg$ is negation,
$\wedge$ is conjunction (and),
$\vee$ is disjunction (or),
$\rightarrow$ is implication and
$\leftrightarrow$ is equivalence.

So, we can give an intuitive meaning to well-formed formulas :

$\forall$_x p(_x)
for every _x , p(_x) is true

$\exists\_x\ (p(\_x, \_y) \wedge q(\_x))$

there exists an _x such that p(_x,_y) is true and q(_x) is true

$\forall\_x\ (\exists\_y\ (q(\_x, \_y) \wedge \neg r(\_y)) \rightarrow p(\_x))$

for every _x, p(_x) is true if there exists an _y such that q(_x,_y) is true and r(_y) is false

If we regard the general forms of program clauses :

$A \leftarrow B_1,...,B_n$ means that for each assignment of each variable, if $B_1,...,B_n$ are all true, then A is true

$A \leftarrow$ means that for each assignment of each variable, A is true

In order to fix the meaning of a logic program, we must also attach a meaning to the constants, the functions and the predicates.

Example : the factorial problem [Kowalski 79]

This is a typical example in conventional programming.

Constant : 0 represents the null integer

Function : S is an unary function S(_x) represents the integer represented by _x incremented by one. So S(0) is 1 , S(S(0)) is 2, ...

Predicates : We suppose that we have a predicate times(_x,_y,_z) which is defined such that it holds when _x times _y is _z

Using program clauses, we can write the following affirmation for the factorial predicate fact(_x,_y) :

fact(0, S(0)) ←
fact(S(_x), _u) ← fact(_x,_v), times(S(_x),_v,_u)

The intuitive meaning of a goal clause is as follows : if $\_x_1$, ... , $\_x_r$ are the variables occuring in

$\leftarrow B_1, ..., B_n$

,the complete notation is

$$\forall \_x_1 \ldots \_x_r (\neg B_1 \vee \ldots \vee \neg B_n)$$

which means that for all $\_x_1$, ... , $\_x_r$, we at least have one B_i (i = 1 ... r) that is false. So, we can equivalently write that there does not exist a combination of $\_x_j$ (j = 1 ... r) such that all B_i are true.

This gives the following formula :

$$\neg \exists \_x_1 \ldots \exists \_x_r (B_1 \wedge \ldots \wedge B_n)$$

Such a clause is used in a refutation demonstration in order to prove whether a combination of values for $\_x_1$, ... , $\_x_r$ exists, combination such that $(B_1 \wedge \ldots \wedge B_n)$, the corresponding goal, is true.

The empty clause is to be understood as contradiction.

A logic programming system must try to see if the set of program clauses completed with the goal clause is inconsistent. Usually, it tries to derive the empty clause by using specific inference rules. If the empty clause is derivable, this means that we have inconsistency. In this case, the system usually gives the bindings, for the variables $\_x_j$ (j = 1 ... r) of the goal clause, which produce the inconsistency. These bindings are so that $(B_1 \wedge \ldots \wedge B_n)$ is true for the values they specify. If such bindings do not exist, the empty clause cannot be derived and it means that $(B_1 \wedge \ldots \wedge B_n)$ can not be true. We also say that $(B_1 \wedge \ldots \wedge B_n)$ can not hold.

Ideally, a logic programming system should be a black box for computing bindings for the variables appearing in the request; the internal workings of the system should be invisible !

1.3.2 Interpretation and truth values :

From a formal point of view, the declarative semantics of a logic program is given by the usual semantics of formulas in first order logic. We briefly recall basic notions ; for a further study, the reader should better consult [Lloyd 84] .

We have seen that a logic program is built using a first order language. The quantifiers and connectives have fixed meanings. This is not the case with constants, functions and predicates. Their meanings are given by an interpretation.

An *interpretation* fixes

- the domain of discourse over which the variables range.
- the assignment of each constant to an element of the domain.
- the assignment of each function to a mapping on the domain.
- the assignment of each predicate to a truth value (true or false) or, equivalently, to a relation on the domain.

The mechanisms to get the truth values for formulas must also be defined.

When a formula expresses a true statement in an interpretation, this latest is called a *model* of the formula.

First order logic provides methods for the deduction of theorems in a first order theory. A *first order theory* consists of an alphabet, a first order language, a set of axioms and a set of inference rules [Mendelson 79] . The formulas are given by the first order language. The axioms are a designated subset of these formulas. In logic programming, the axioms are the program clauses.

The *theorems* are in fact the formulas coming as logical consequences of the axioms. This means they are true for any interpretation which is a model of all the axioms.

The inference rule used by logic programming systems under consideration is the resolution principle introduced by Robinson in 1965 [Robinson 65].

1.3.3 Substitution, unification, answer substitution :

In automatic theorem proving, it suffices to demonstrate logical consequence but in logic programming, the aim is to compute bindings for variables appearing in a formula. These bindings are such that, when we replace the variables of the formula with the values they specify, the formula becomes a logical consequence of the axioms (or equivalently of the program). These bindings are the output from the running of the program. A declarative understanding of the output of a program and a goal is given by the concept of correct answer substitution.

A *substitution* θ is a finite set of the form $\{\_v_1/t_1 , \ldots , \_v_n/t_n\}$ where each ti is a term distinct from $\_v_i$ and the variables $\_v_1, \ldots , \_v_n$ are distinct. Each element $\_v_i/t_i$ is called a binding for $\_v_i$. When all the t_i are ground terms, we have a ground substitution (a ground term being a term not containing variables). When all the t_i are variables, we have a variable pure substitution.

What we will call an *expression* is either a term, a literal or a conjunction or disjunction of literals. A simple expression is either a term or an atomic formula.

If $\theta = \{\_v_1/t_1 , \ldots , \_v_n/t_n\}$ is a substitution and E an expression, then $E\theta$ is the *instance* of E by θ . It is obtained by simultaneously replacing each occurrence of the variable $\_v_i$ in E by the term t_i (i = 1 ... n). If $E\theta$ is ground, then $E\theta$ is called a ground instance of E.

Let $\theta = \{\_u_1/s_1, \ldots , \_u_m/s_m\}$ and $\sigma = \{\_v_1/t_1, \ldots , \_v_n/t_n\}$ be substitutions. The *composition* $\theta\sigma$ of θ and σ is the substitution obtained from the set $\{\_u_1/s_1\sigma , \ldots , \_u_m/s_m\sigma , \_v_1/t_1, \ldots , \_v_n/t_n\}$ by deleting any binding $\_u_i/s_i\sigma$ such that $\_u_i = s_i\sigma$ and deleting any binding v_j/t_j such that $v_j \in \{\_u_1, \ldots , \_u_m\}$.

Example :

$\theta = \{\_x/f(\_y), \_y/\_z\}$, $\sigma = \{\_x/a, \_y/b, \_z/\_y\}$. Their composition $\theta\sigma$ is $\{\_x/f(b), \_z/\_y\}$ and their composition $\sigma\theta$ is $\{\_x/a, \_y/b\}$

The substitution given by the empty set is called the identity substitution and is denoted by ε .

We can list the following properties :

if θ, σ and γ are substitutions and E an expression then

- $\theta\varepsilon = \varepsilon\theta = \theta$
- $(E\theta)\sigma = E(\theta\sigma)$
- $(\theta\sigma)\gamma = \theta(\sigma\gamma)$

If E and F are expressions, they are called *variants* if there exist substitutions θ and σ such that $E = F\sigma$ and $F = E\theta$. It is also said that E (F) is a variant of F

(E).

Examples :

p(_x,_y) and p(_v,_w) are variants
q(a) and q(_x) are not variants
r(_x,_x) and r(_x,_y) are not variants

Assuming that E is an expression and V is the set of all the variables occurring in E, a renaming substitution is a variable-pure substitution $\{\_v_1/\_t_1, \ \dots \ , \ \_v_n/\_t_n\}$ such that $\{\_v_1, \dots, \_v_n\} \subseteq V$, the $\_v_i$ are distinct and $(V \setminus \{\_v_1, \dots , \_v_n\}) \cap \{\_t_1, \dots , \_t_n\} = \varnothing$.

It has been proved that for any of both variants E and F, there exists a renaming substitution θ (σ) for E (F) such that $F = E\theta$ ($E = F\sigma$).

The concept of *unifier* is fundamental. Unification was first introduced by Herbrand and is much used in logic programming systems. The idea is to find a substitution for a set of expressions, substitution such that the set of instances of these expressions by the substitution is a singleton (thus, all instances are equivalent) . For our purpose it is enough to consider non-empty finite sets of simple expressions (terms or atomic formulas).

If S is a set of expressions of the form $\{E_1, \ \dots \ , E_n\}$ and a substitution, we write $S\theta$ for $\{E_1\theta \ , \dots , E_n\theta\}$.

If $S\theta$ is a singleton, then θ is a unifier for S. It is also said that E_i (i = 1...n) match together via θ , which is the matching substitution.

A unifier θ for S is called a most general unifier (MGU) if for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.

A unification algorithm (providing occur check) is presented in [Lloyd 84]. The occur check is fundamental but has major drawbacks on efficiency. We can illustrate the occur check with the following set of expression $S = \{\_x, f(\_x)\}$.

For algorithms performing occur check, S is not unifiable since _x appears in f(_x) but other algorithms will unify _x and f(_x) and this can cause problems as we will see later.

If we say that an answer substitution for $P \cup \{\leftarrow G\}$ is a substitution for variables appearing in G (not necessarily all variables) , we can provide the following definition of *correct answer substitution* :

let P be a program,
let G be a goal of the form ($B_1, \ldots, B_n$) and
let θ be an answer substitution for $P \cup \{\leftarrow G\}$

θ is a correct answer substitution for $P \cup \{\leftarrow G\}$ if $\forall ((B1 \wedge \ldots \wedge Bn)\theta)$ is a logical consequence of P .

This concept captures the intuitive meaning of "correct answer". It provides a declarative understanding of the desired output of a program and a goal.

A logic programming system should also return the answer "no" if $P \cup \{\leftarrow G\}$ does not lead to inconsistency (thus in the case there is no correct answer substitution).

1.4 PROCEDURAL SEMANTICS :

We now turn to the most usual procedural interpretation of Horn clauses. This interpretation makes Horn clause logic very effective as a programming language.

We have seen at the declarative level that the aim of logic programming is to compute correct answer substitutions. At the procedural level the counterpart of this concept is the concept of computed answer substitution which is defined by using a refutation procedure called SLD-resolution. Hereafter, we take over from Lloyd [Lloyd 84] the basic definitions and results of some interest for our purpose.

1.4.1 SLD-resolution :

A *computation rule* is a function from a set of goals to a set of atomic formulas (or atom), such that the value of the function for a goal is always an atom, called the selected atom, in that goal.

Let GC_i be $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and C_{i+1} be $A \leftarrow B_1, \ldots, B_q$ and R be a computation rule. then GC_{i+1} is derived from GC_i and C_{i+1} using MGU θ_{i+1} via R if

the following conditions hold :

- A_m is the selected atom given by the computation rule R
- $A_m \theta_{i+1} = A \theta_{i+1}$ (θ_{i+1} is an MGU of A_m and A)
- GC_{i+1} is the goal clause $\leftarrow(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta_{i+1}$

GC_{i+1} is called a resolvent of GC_i and C_{i+1} .

Let P be a program, G a goal and R a computation rule. An *SLD-derivation* of P $\cup$ {$\leftarrow$G} via R consists of a (finite or infinite) sequence $GC_0 = \leftarrow G$, GC_1, ... of goal clauses, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P and a sequence θ_1, $\theta_2, \ldots$ of MGU's such that each GC_{i+1} is derived from GC_i and C_{i+1} using θ_{i+1} via R.

Each C_i is a suitable variant of the corresponding program clause so that C_i does not have any variables which already appear in the derivation up to GC_{i-1}. This can be achieved, for example, by subscripting variables in G by 0 and in C_i by i. This process of renaming variables is called *standardizing* variables apart . It is necessary, otherwise, for example, we would not be able to unify p(_x) and p(f(_x)) in $\leftarrow$p(_x) and p(f(_x)) $\leftarrow$. Each C_i is called an *input clause* of the derivation.

An *SLD-refutation* of P $\cup$ {$\leftarrow$G} via R is a finite SLD-derivation of P $\cup$ {$\leftarrow$G} via R which has the empty clause as the last goal in the derivation.

SLD-derivations can be finite or infinite. A finite SLD- derivation can be successful or failed. A successful SLD- derivation is one that ends in the empty clause; so it is a refutation. A failed SLD-derivation is one that ends in a non-empty goal with the property that the selected atom in this goal does not unify with the head of any program clause.

Now, we can give a definition for the concept of *computed answer substitution* (CAS) :

> an R-computed answer substitution for P $\cup$ {$\leftarrow$G} is the substitution obtained by restricting the composition $\theta_1 \ldots \theta_n$ to the variables

of G, where $\theta_1, \ldots, \theta_n$ is the sequence of MGU's used in an SLD-refutation of $P \cup \{\leftarrow G\}$ via R .

It has been proved by Clark [Clark 79] that SLD-resolution is sound and complete.

Soundness :

this means that if we consider a program P, a goal G and a computation rule R, every R-computed answer substitution for $P \cup \{\leftarrow G\}$ is a correct answer substitution.

Completeness :

this means that if we consider a program P, a goal G and a computation rule R, for every correct answer substitution σ for $P \cup \{\leftarrow G\}$, there exists an R-computed answer substitution θ for $P \cup \{\leftarrow G\}$ and a substitution γ such that $\sigma = \theta\gamma$.

We can also say that every computed answer substitution is correct and that every correct answer substitution is an instance of a computed answer substitution. The equivalence of the two semantics is the core of logic programming. A fundamental consequence of this equivalence is that to write a logic program, we can reason in term of logic; no procedural aspects should intervene in the construction process. Therefore, a logic program can be seen as the description of a problem in logic. Moreover, given the problem is described in term of relations (predicates), the logic program can be used whatever the instantiation of the parameters is (some arguments being input datas while the others are output results). This is called multidirectionnality [Deville 87] .

1.4.2 SLD-refutation procedures :

Now, there are many strategies a system may adopt in its search for a refutation, the search space being a certain type of tree, called a SLD-tree. The results herebefore allow the building of the SLD-tree using one computation rule fixed by advance.

Here follows the definition of a SLD-tree !

Let P be a program, G a goal and R a computation rule ; then the *SLD-tree* for P

$\cup$ $\{\leftarrow G\}$ via R is defined as follows :

- each node of the tree is a goal clause (possibly empty) .
- the root node is $\leftarrow G$.
- let $\leftarrow A_1,...,A_m,...,A_k$ $(k\geq 1)$ be a node in the tree and suppose that A_m is the atom selected by R. Then this node has a descendent for each input clause $A\leftarrow B_1,...,B_q$ such that A_m and A are unifiable. The descendent is $\leftarrow(A_1,...,A_{m-1},B_1,...,B_q,A_{m+1},...,A_k)\theta$ where θ is the MGU of A_m and A.
- Nodes which are the empty clause have no descendents.

Each branch of the SLD-tree is a derivation of P $\cup$ $\{\leftarrow G\}$. Branches corresponding to successful derivations are called *success branches*, branches corresponding to infinite derivations are called *infinite branches* and branches corresponding to failed derivations are called *failure branches.*

The next example provides illustration for these concepts :

let P be the following program :

p(_x,_z) ← q(_x,_y), p(_y,_z)
p(_x,_x) ←
q(a,b) ←

let G be the following goal :

← p(_x, b)

The first tree (figure 1.1) is built using the computation rule that selects the leftmost atom of the goal, the second (figure 1.2) with the computation rule that selects the rightmost atom of the goal. The first tree is finite while the second is infinite but both have two success branches corresponding to the answers {_x/a} and {_x/b}.

The annotations of the arcs are the MGUs used.

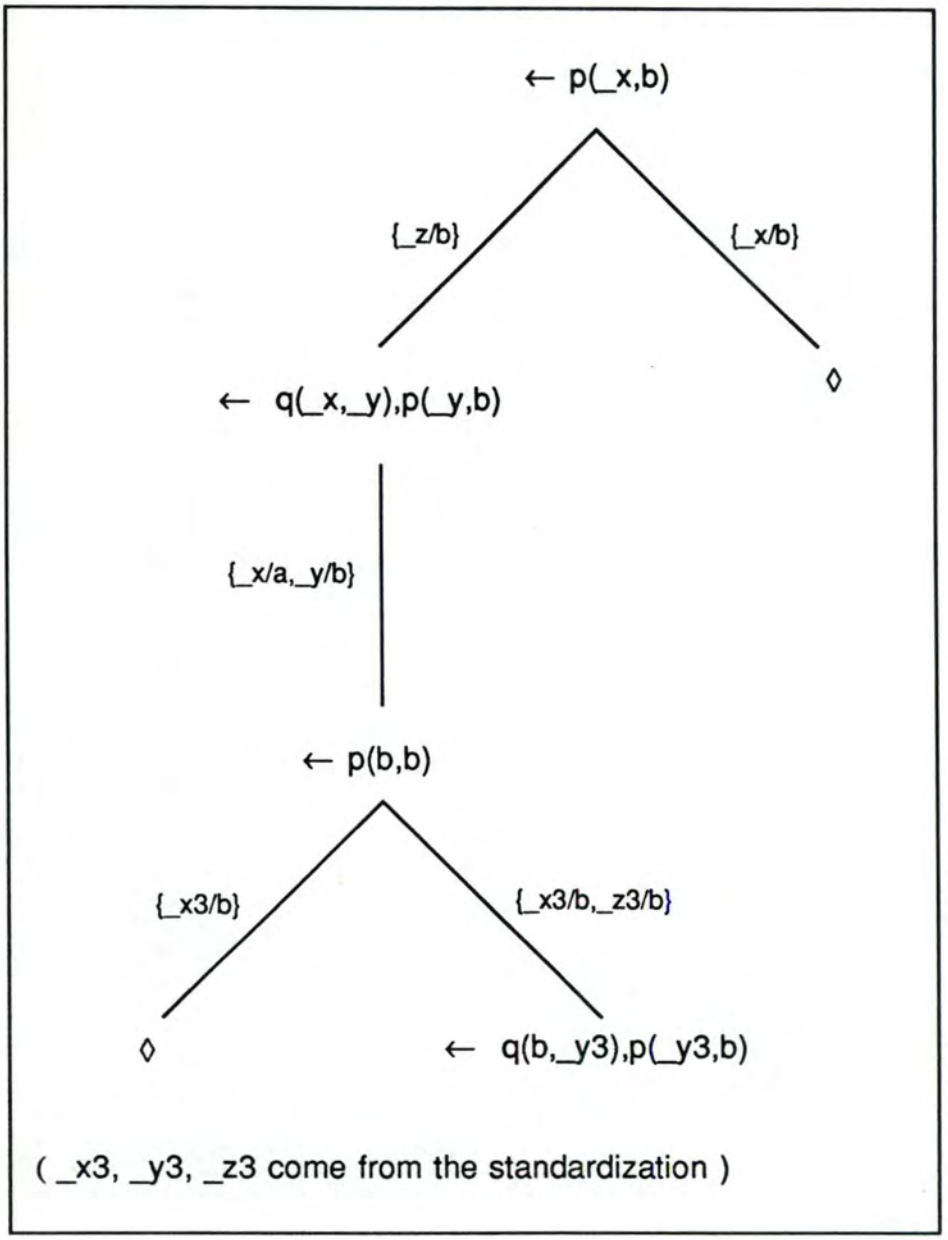
← p(_x,b)
{_z/b}
{_x/b}
◊
← q(_x,_y),p(_y,b)
{_x/a,_y/b}
← p(b,b)
{_x3/b}
{_x3/b,_z3/b}
◊
← q(b,_y3),p(_y3,b)
(_x3, _y3, _z3 come from the standardization)

Figure 1.1

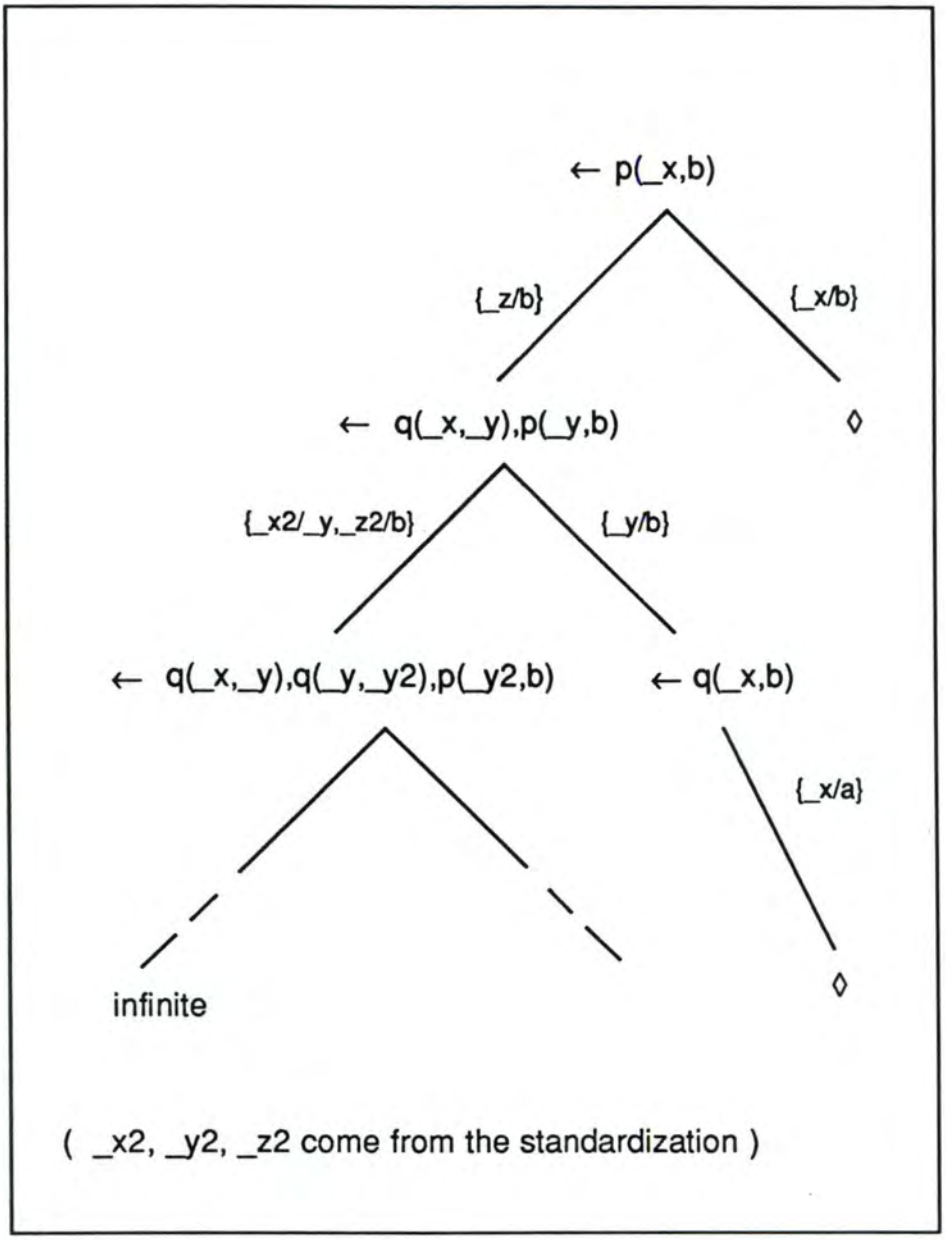
← p(_x,b)
{_z/b}
{_x/b}
← q(_x,_y),p(_y,b)
◊
{_x2/_y,_z2/b}
{_y/b}
← q(_x,_y),q(_y,_y2),p(_y2,b)
← q(_x,b)
{_x/a}
infinite
◊
(_x2, _y2, _z2 come from the standardization)

Figure 1.2

By convention, we denote T(G) the SLD-tree for P $\cup$ {$\leftarrow$G} when there is no ambiguity over the program P and the computation rule R !

A last thing has to be fixed, it is the *strategy* the system uses for searching SLD-trees. To define this strategy, we must introduce an order between the descendents of a node. So, we can speak of a *sequence of descendents*. Now, there are many kinds of search the system might adopt to find success branches and thus the corresponding computed answer substitutions. The depth-first search, by instance, fully explores the branches passing by the i^{th} descendent of a sequence before it explores those passing by the $(i+1)^{th}$. This strategy is also called the *search rule*. In the next section the importance of this choice is underlined.

We call the *i^{th} descendent input clause* of a node α the input clause that permits to get the i^{th} descendent of α .

When a descendent d_1 appears before a descendent d_2 in the sequence of descendents, we say that d_1 is more at left than d_2 (similarily, d_2 is more at right than d_1). This denominationnal convention is inspired from the graphical representation of trees when the i^{th} descendent of a node α is displayed using the ith leftmost bound starting at node α. We also say that a branch B_1 is located more at left than a branch B_2 if α being their last common node, B_1 passes by a descendent, of α, more at left than the one included in B_2.

The combination of a computation rule with a search rule defines an *SLD-refutation procedure.*

But we must not forget the semi-decidability of first order logic which is a first major drawback. The problem is that when a goal is not a logical consequence of the logic program, the execution of an SLD-refutation procedure may never terminate. As non-termination is an incorrect behaviour, some programs, correct at the declarative level (this means being a good translation of the problem specification in terms of logic) , can be unacceptable from a procedural point of view and this does not depend on the logic programming system.

1.5 PROLOG :

PROLOG is one of the first attempts that have been made to provide a programming language based on logic and more specifically on Horn clause logic. It was introduced mainly by Colmerauer whose team built the first interpreter [Colmerauer 73].

In this work, we refer to standard PROLOG (unless explicitly said different). A representative choice for it is the PROLOG language described in by Clocksin and Mellish [Clocksin 81]. In most of our examples, we use its syntax and built-in procedures. Note however that variables still begin by an underscore rather than a capital letter. This corresponds to the BIM_prolog syntax [BIM 86] .

It is assumed that the reader is familiar with PROLOG and also with the tricky concept of *backtracking*. Let us recall that, in PROLOG, the connectives $\neg$ and $\leftarrow$ are respectively written as ?- and :-. Remember also that PROLOG rules, unlike Horn clauses, end by a point.

Working with PROLOG, a programmer should ideally solve a problem by building a description of it under the form of Horn clauses, so working at the declarative level. But this ideal is far from beeing reached. In fact, there are many restrictions to the logical aspect of PROLOG. We mention some of them hereafter, relying on [Deville 87] and [Lloyd 84].

Let us first recall that the PROLOG interpreters are performing an SLD-refutation procedure with selection of the leftmost atom as computation rule. The search rule of PROLOG is to perform a depth first search on SLD-trees where the descendents are ordered so that the i^{th} descendent, of a node α, can be obtained by using the i^{th} descendent input clause which is a suitable variant of the i^{th} program clause for which the head of a variant is unifiable with the first subgoal of α . PROLOG also provides the possibility of having negated literals in the body of a clause. This proves useful in practice. Negation is handled by a failure rule: the idea is to derive $\neg Q$ if it is impossible to derive Q from the logic program.

PROLOG also makes possible to include extra-logical features in program clauses :

- control information : the cut (!) allows to prune the SLD-tree.
- input-output primitives : these are necessary; they produce input-output by side-effects.

- assignation mechanism : this is one of the most (if not the most) used operation in computer science. In PROLOG, it is implemented as a side effect of the *is* predicate.
- others : *bagof, setof, assert, retract,* ... are classical examples.

These primitives are outside the scope of first-order logic but can be useful in practice.

The use of cuts can have some desastrous consequences. Given the pruning, misplaced cuts can lead to incompleteness. Though it is written in the body of clauses like an atom, the cut has no logical significance but it is handy to see it as an atom that immediately succeeds at the first call and fails on backtracking, pruning the SLD-tree in the same time. If the pruned part contains an answer, then the cut is unsafe (by opposition to safe) and we get incompleteness. Programmers can also use cuts to write program which are not even declaratively correct !

To illustrate this matter, consider the following example from [Lloyd 84] :

```
max(_x,_y,_y) :- _x <= _y, ! .
max(_x,_y,_x).
```

max(_x,_y,_z) is intended to be true if and only if _z is the maximum of _x and _y. Procedurally, the semantics of the above program is the maximum relation but declaratively, it is something else entirely.

The only advantages of the cut are at efficiency level.

The negation by failure rule is also acceptable from the efficiency point of view, but in PROLOG, it can lead to unsoundness and incompleteness.

Example of unsoundness :

```
p(a).
r(_y) :- not( p(_y) ) .
```

The goal clause ?- r(b). succeeds though r(b) is not a logical consequence of this logic program.

Example of incompleteness :

```
q(a) :- not( r(a) ) .
q(a) :- r(a).
```

```
r(_x) :- r( f(_x) ).
```

We obviously have that q(a) is a logical consequence [Lloyd 84]. Though, there cannot be a succesful derivation for q(a). In order to derive $\neg$r(a), negation as failure tries to show the impossibility of deriving r(a). But here, it has to search an infinite SLD-tree. This is also a consequence of the semi- decidability of first-order logic [Deville 87]. This incompleteness is independent of the computation and search rules.

The PROLOG search rule also leads to incompleteness. The following example is an illustration of the problem [Lloyd 84]:

```
p(a,b) .
p(c,b) .
p(_x,_z) :- p(_x,_y) , p(_y,_z) .
p(_x,_y) :- p(_y,_x) .
```

p(a,c) is a logical consequence of the program but PROLOG will never find a refutation for the goal clause ?- p(a,c). because the left most branch of the corresponding SLD-tree is infinite.

Another problem of PROLOG systems is the unification algorithm used which not always performs the occur check for efficiency reasons. Without occur check, it is possible to unify _x and f(_x) and so, to produce an infinite term f(f(f(...))) . Normally, they should not unify ! Such a fault can lead to unsoundness as the example hereafter shows [Lloyd 84] :

```
test :- p(_x, _x) .
p(_y, f(_y)) ;
```

A PROLOG system without occur check would give the answer substitution e for the goal (test). It is thus considered as a logical consequence of the program but it is quite wrong ! To avoid this problem, the unification algorithm must provide occur check. In the rest of the work, it is assumed to be so.

However, in conclusion, we can claim that programming in PROLOG is not the same as programming in logic [Deville 87] and that much characteristics of PROLOG have to be explained outside the scope of the logic programming paradigm.

CHAPTER 2 : A NEW PROCEDURAL SEMANTICS FOR PROLOG

2.1 INTRODUCTION :

In chapter 1, we have seen that some PROLOG features produce a gap between the procedural semantics and the declarative semantics. So, programmers cannot simply continue reasoning in logic for building a PROLOG program.

A possible solution is to provide to programmers a methodology that tries to reconciliate the two semantics so that the logic programming paradigm can be used in the construction process of a PROLOG program. Such a methodology has been deleloped in [Deville 87]. The basic idea is to construct a logic algorithm in pure logic from the specifications, independently of any programming language or procedural semantics. Then, the use of derivation rules enables to get a logic program in PROLOG from the logic algorithm.

Our work is humbler and tries to give an accurate but simple (we hope so) way of specification of the procedural semantics of PROLOG programs. We think it could be useful because in order to achieve program correctness in PROLOG, progammers need to reach an accurate knowledge of its procedural aspects. But the problem is the difficulty of building correctness proofs using the usual procedural semantics of logic programs. Moreover, this semantics can be considered as "not suited" because of the PROLOG features coming out of its scope.

The procedural semantics for PROLOG programs that we propose should normally deal with all PROLOG particularities and is much inspired of the usually intuitive (and informal) understanding of the execution mechanisms of the language.

So, we begin by a quick recall over this intuitive understanding. Then, we present some fundamental concepts needed by the three algorithms ESG (Execute SubGoal), EG (Execute Goal) and EQ (Execute Question) which are the core of our work.

Note that for the moment, the only extra-logical feature we take into account is the cut !

2.2 USUAL INTUITIVE UNDERSTANDING :

The usual intuitive understanding of the procedural interpretation of PROLOG programs is much inspired from the classical problem-reduction (top-down problem solving) strategy developed in artificial intelligence.

Consider the goal clause

$$?- A_1 , \ldots , A_m .$$

with A_i (i = 1 ... m) atomic formulas and $\_x_j$ (j = 1 ... n) the variables appearing in it. It can be interpreted as follows :

try to make the goal (A_1 , ... , Am) hold

or, more explicitly,

try to find $\_x_1, \ldots , \_x_n$ which make the goal (A_1 , ... , A_m) hold.

This operation, when it succeeds, provides a substitution θ such that (A_1 , ... , A_m)θ is true.

A program clause

$$A :- A_1, ..., A_m .$$

with A, A_i (i = 1 ... m) atomic formulas, can be interpreted as an entry point of a procedure (a procedure being defined by all the program clauses with same functor in head). This part of the procedure means :

to make a subgoal of the form of A hold,
try to make the goal (A_1, ... , A_m) hold.

Given a subgoal B which matches A via a substitution θ, some combination of values (for the variables appearing in B) that make B hold can be found by using θ and the combinations of values that make ($A_1 \theta$, ... , $A_m \theta$) hold .

A unit clause

A .

can be also interpreted as an entry point of a procedure but this part of the procedure makes the subgoals matching A hold without reducing them to further subgoals.

If it is impossible to make a subgoal or a goal hold, it is said to fail !

Trying to make a goal $(A_1 , ... , A_m)$ hold can be decomposed into m steps :

- try to make A_1 hold .
- try to make $A_2\ \theta_1$ hold .
- try to make $A_3(\ \theta_1\ \theta_2)$ hold .
- ...
- try to make $A_m(\ \theta_1 ...\ \theta_{m-1})$ hold .

where θ_i (i = 1 ... m) is a substitution defining values for the variables appearing in $A_i(\ \theta_1 ...\ \theta_{i-1})$, values such that it holds. When a subgoal $A_i(\ \theta_1 ...\ \theta_{i-1})$ fails (i = 1 ... m), backtracking occurs. If i = 1 , then the goal $(A_1, ... , A_m)$ fails. If i > 1, it means to go back to the previous step and to try to make $A_{i-1}(\ \theta_1 ... \theta_{i-2})$ hold but with a new combination of values. So, it means the progression through the different steps is not purely sequential ! When a solution has been given, it means all steps have succeeded. In order to find another combination of values such that the goal holds, the idea is to act as in backtracking but going back to the mth step.

Trying to make a subgoal hold can also be subdivised in steps. There will be n possible steps if there are n program clauses whose heads match the subgoal. To each step corresponds one of these clauses. The different steps are ranked following the order of appearance in the program of their corresponding clauses. If there are n such clauses, it means that, for the subgoal at hand, there are n opportunities of entry point to the procedure. If $(A_{i\ 1} ...\ A_{i\ mi})$ is the body of the i^{th} matching clause and θ_i is the matching substitution, the n steps can be described as follows :

- try to make the goal $(A_{1\,1}, \ldots, A_{1\,m1})\,\theta_1$ hold .
- ...
- try to make the goal $(A_{n\,1}, \ldots, A_{n\,mn})\,\theta_n$ hold .

The progression between these steps is as follows : when all possible combinations of values making $(A_{i\,1}, \ldots, A_{i\,mi})\,\theta_i$ hold have been found (or equivalently when this goal fails), the next combination of values for the subgoal variables is searched using the next step, thus i+1 (if i = n the subgoal fails) .

Now, we can see that it is far from being clear and simple. Moreover, we have not explained yet the effects of a cut and backtracking has only been considered in its very local consequences !

We will not try here to explain the full consequences of backtracking because from our point of view, it goes far beyond an informal presentation. It might be one of the reason why many PROLOG programs are filled with cuts. Novice PROLOG programmers often abuse of cuts to ensure themselves that no unwanted and un- understandable backtracking will occur.

The cut has in general more localized effects and their descriptions are very often given in an informal way. Consider the following goal :

$$(A_1, \ldots, A_{i-1}, !, A_{i+1}, \ldots, A_m) .$$

The cut can be seen as a subgoal which holds immediately at the first attempt and fails for any other attempt to make it hold (thus, on backtracking). But the cut has side effects: when it fails, the whole goal, in which it is, fails immediately and so, it is impossible to backtrack on A_j (j = 1 ... i-1) when the cut has been passed. And when the cut is encountered while trying to make a subgoal hold, or more precisely when trying to make the body of a matching clause hold, then it has for effect that all the following matching clauses will never be considered.

Our conviction is that the usual intuitive understanding presented here is not well-suited for program correctness proofs or program construction and is not always easy to "master". We think it is mostly due to its lack of accuracy and clarity. In the following sections, we introduce a definition of the procedural semantics which is inspired from the informal interpretation explained herebefore but tries to be accurate, complete and, above all, as simple as possible.

2.3 PRELIMINARIES :

In this section, we present definitions and properties of some fundamental concepts for our expression of the procedural semantics.

2.3.1 Context :

For the moment, it is enough to consider that a *context* is characterized by a program. Later, when introducing input-output built-in procedures, its characterization will also include the set of accessible files.

2.3.2 Array of contextual variables (ACV) :

An *array of contextual variables* (ACV) is a finite array of elements $\_v_i/t_i$ (i = 1 ... n) where the variables $\_v_1$, ..., $\_v_n$ are distinct and where each t_i is a term. So, an ACV can be written as follows :

$$[\_v_1/t_1 , \ldots , \_v_n/t_n] .$$

Note : the brackets have not the same meaning than in PROLOG !

Each element $\_v_i/t_i$ is called a *binding* for $\_v_i$. If V is a ACV, it is called a ground ACV if the t_i are all ground terms and a variable-pure ACV if the t_i are all variables.

This notion is of course very similar to the one of substitution. The main differences is that in the ACV $[\_v_1/t_1 , \ldots , \_v_n/t_n]$, we do not require that $t_i \neq \_v_i$ (i = 1 ... n) . We introduce it because when trying to understand a PROLOG program, a programmer usually reasons with a set of variables which are progressively instantiated rather than in terms of substitution compositions.

In our algorithms, we use variables whose value is an ACV. We call these variables VACV (Variable of type Array of Contextual Variables).

Note : the variables we use in our algorithm are classical variables as in Pascal language for example.

A VACV being a variable, it can be assigned and its value can evolve in time .

We need this notion because we want to consider the evolution of the value for the variables having a binding in the array ; this captures the progressive instantiation related in the previous paragraph.

So, if we adopt a pseudo-pascal syntax, assuming that V and W are VACVs, we can write :

$V := [\_v_1/t_1 , \_v_2/t_2]$; { assignation of a value to V }
$W := [\_v_3/t_3 , \_v_4/t_4]$; { assignation of a value to w }
V := W ; { V takes the value of W ; so after execution of this instruction, $V = [\_v_3/t_3 , \_v_4/t_4]$ }

By convention, a VACV which has never been assigned, has the empty array ([]) for value.

Note however that we do not often make the distinction between an ACV and a VACV, considering VACVs as ACVs. In fact, definitions and properties stated for ACVs can easily (and sometimes immediately) be extended to VACVs .

A. Substitution defined by an ACV :

Let V be an ACV .

If $V = [\_v_1/t_1, \ldots , \_v_n/t_n]$ ($n \geq 0$), ΘV denotes the substitution defined by V and it is obtained from the set $\{\_v_1/t_1 , \ldots , \_v_n/t_n\}$ by deleting any binding $\_v_i/t_i$ for which $\_v_i = t_i$ ($i = 1 \ldots n$) .

B. Instance of an ACV by a substitution :

Let V be an ACV, let $\theta = \{\_y_1/s_1, \ldots, \_y_m/s_m\}$ be a substitution

$V\theta$ denotes the instance of V by θ . If $V = [\_v_1/t_1 , \ldots , \_v_n/t_n]$ then, $V\theta = [\_v_1/(t_1\theta) , \ldots , \_v_n/(t_n\theta)]$

Of course, the substitution can be a substitution defined by a ACV.

Note that when we speak about an instance of an expression by an ACV, the meaning is the instance of the expression by the substitution defined by an ACV.

C. Restriction of a substitution by an ACV :

Let θ be a substitution and V be an ACV .

$< \theta >V$ denotes the restriction of the substitution θ by the ACV V. This restriction is the substitution obtained from θ by deleting any binding for variables which do not appear in V .

We say a variable *appears* in V either if it is one of the $\_v_i$ or if it appears in at least one t_i.

Example :

$\theta = \{\_x_1/t_1 , \_x_2/t_2 , \_x_3/t_3\}$
$V = [\_x_1/s_1 , \_y/\_x_2]$

$< \theta >V = \{\_x_1/t_1 , \_x_2/t_2\}$

D. Strict restriction of a substitution by an ACV :

Let θ be a substitution and V be an ACV.

$« \theta »V$ denotes the strict restriction of the substitution θ by the ACV V. This restriction is the substitution obtained from by deleting any binding for variables there is no binding for in V .

Example :

$\theta = \{\_x_1/t_1 , \_x_2/t_2 , \_x_3/t_3\}$
$V = [\_x_1/s_1 , \_y/\_x_2]$

$« \theta »V = \{\_x_1/t_1\}$

E. Proposition 2.1 :

Proposition :

let θ be a substitution and V an ACV

if $\Theta V = \varepsilon$ then $< \theta > V = « \theta » V$.

Proof :

this comes immediately from the fact that if $\Theta V = \varepsilon$, each binding of V is of the form _v/_v and so, each variable appearing in V is also concerned by a binding in V .

qed

F. Proposition 2.2 :

Proposition :

let V and V1 be ACVs
let $\theta = \{\_y_1/s_1 , \ldots , \_y_m/s_m\}$ be a substitution

if $V = [\_x_1/t_1 , \ldots , \_x_n/t_n]$ and if $V_1 = V\theta$,

then ΘV_1 is equal to the substitution (ΘV θ) strictly restricted by V, so $\Theta V_1 = \Theta(V\theta) = « \Theta V \ \theta » V$.

Proof :

we can write that $V_1 = [\_x_1/ t_1 \theta , \ldots , \_x_n/ t_n \theta]$ and
$\Theta V_1 = \{\_x_i/ t_i \theta , 1 \leq i \leq n : \_x_i \neq t_i \theta \}$

We proceed in three steps !

Step 1 :

$\_x_i/ t_i \theta \notin \Theta V1 \Rightarrow \neg\exists \ \_x_i/t'_i \in (\Theta V \theta)$ $(i = 1 \ldots n)$

As $\_x_i/t_i \theta \notin \Theta V_1$, we know that $\_x_i = t_i \theta$. We consider two situations : $\_x_i = t_i$ and $\_x_i \neq t_i$.

1> $\_x_i = t_i$:

this means that $\_x_i / t_i \notin \Theta V$. Therefore, it is only possible to have a binding $\_x_i / t'_i \in (\Theta V \theta)$ if there is a binding for $\_x_i$ in θ or, equivalently, if $\exists\ j\ (1 \le j \le m) : \_y_j = \_x_i$ and $s_j \neq \_x_i$. This is impossible because we would have

$t_i \theta = \_x_i \theta = s_j$ with $s_j \neq \_x_i$ but by hypothesis, $\_x_i = t_i \theta$.

2> $\_x_i \neq t_i$:

this means that $\_x_i / t_i \in \Theta V$. So, by definition of substitution composition, the binding $\_x_i / t'_i \in (\Theta V \theta)$ can only be of the form $\_x_i / t_i \theta$; but such a binding is not retained in $(\Theta V \theta)$ as $\_x_i = t_i \theta$.

Step 2 :

$$\_x_i / t_i \ \theta \in \Theta V_1 \Rightarrow \_x_i / t_i \theta \in (\Theta V \theta) \quad (i = 1 \ldots n)$$

As $\_x_i / t_i \theta \in \Theta V_1$, we know that $\_x_i \neq t_i \theta$. So, due to the definitions of substitution composition and instantiation of an ACV, we have immediately that $\_x_i / t_i \theta \in (\Theta V \theta)$.

Step 3 :

Using modus tollens on the proposition of step 1 and, then, applying proposition of step 2, we get

$$\_x_i / t'_i \in (\Theta V \theta) \Rightarrow t'_i = t_i \theta \quad (I)$$

So, the thesis follows immediately from step 2, (I) and the fact that the bindings in θ for variables not concerned by any binding of V do not appear in the instance of V by θ .

qed

G. Covering :

If E is an expression, we say that the ACV V covers E if for any variable _v appearing in E, a binding for this variable exists in V .

2.3.3 General primitives :

As the description of the procedural semantics is made using algorithms, it is handy to assume the existence of some general primitives. This clarifies the algorithms.

A. Function REFERENCE_ACV(CL) :

This primitive unary function receives as input a clause and returns an ACV which contains a binding _v/_v for each variable _v appearing in the clause received in input.

Example :

if CL is a clause of the form

p(_x, 5) :- q(_x, _y), q(_y, 5). ,

the instruction

V := REFERENCE_ACV(CL) ;

where V is an VACV, has for effect that after execution,

V = [_x/_x , _y/_y]

B. Function STANDARDIZATION(CL, V) :

This function receives as input a clause CL and an VACV V and returns a variant clause of CL such that it does not contain any variable appearing in V.

Note that this standardization is not as strong as the one presented in the definition of SLD-derivation (section 1.4) but it is enough for our version of the procedural semantics as it will be proved.

C. Function MGU(CL, SG, V) :

This function receives as input a clause CL, a subgoal SG and an VACV V and if the head of CL and SG are unifiable, it returns a substitution which is an MGU for them. The MGU returned is so that it does not contain any variable appearing in V but not in SG.

Example :

let V = [_x/f(_y) , _z/_z]
let CL be p(_w) :- q(_w,6).
let SG be p(_z).

if we execute the call MGU(CL, SG, V) , {_z/_v , _w/_v} is a possible result of it, but {_z/_y, _w/_y} is not because _y appears in V.

This definition of function MGU avoids unwanted links that could occur between variables !

Example :

consider the previous example. If we accept {_z/_y , _w/_y}, as V = [_x/f(_y) , _z/_z] , we introduce a link between _x and _z without any justification.

2.3.4 Special mechanisms :

We must explain the execution mechanisms of the algorithms EG and ESG because they are fundamental for the understanding of our procedural semantics and are not conventionnal. In this section, we introduce the specific concepts.

The text of these algorithms cannot be executed immediately as the text of a common procedure. Before execution, a creation of an incarnation is needed. An incarnation of an algorithm is in fact a specific instance of it. It is then possible to have many incarnation of the same algorithm and even, some incarnations can be similar to all respect excepted for their respective stamps. A *stamp* is a unique value which is attributed to an incarnation at its creation and which is kept until the destruction of the incarnation.

It is at creation that the calling environment must specify the actual parameters once for all. The link between the formal and actual parameters is done like for a call by address if (like in Pascal) a "var" appears before the declaration of the formal

parameter and like a call by value otherwise. This link remains for all the life of the incarnation created.

The creation is made by a call to a specific procedure "create" and has the following generic form :

create(algo_name(parameters)) .

Destruction of an incarnation is in fact self-destruction. It is performed by calling the procedure "selfkill" which makes the execution of the incarnation stop and destroys it. The procedure "selfkill" has no parameters.

During its lifetime, an incarnation can be executed by a call to the procedure "execute". A call to "execute" has the following generic form :

execute(incarn_stamp)

where incarn_stamp is the stamp of the incarnation to be executed.

The termination of an execution is done by performing a call to the procedure "terminate". This procedure takes one parameter which is the name of a label (usual meaning) appearing in the algorithm corresponding to the incarnation being executed. With "terminate", the current execution is ended but the incarnation remains in life. The label name is in fact memorized in a special remanent variable : *entry_pt* . This variable is used at the start of an execution to determine where to branch. There are other remanent variables, depending on the algorithm which is incarnated. They will be introduced in the presentation of each algorithm.

Some remanent variables are initialized by the creation operation. The value of initialization for these variables is specified with their declaration. The generic forms for the declaration of a remanent variable are as follows :

<variable_name> : <variable_type>

or

<variable_name> : <variable_type> / init := <initialization_value>

We end this section with some notational conventions over goals. Our algorithms treat goals as lists of subgoals. The empty goal is denoted (). If a goal G is composed of a first goal SG followed by a list of subgoals G' then, we can write G = (SG,G') but also G = SG,G' .

2.4 ALGORITHMS :

We come here in the core of our work : the expression of the procedural semantics with the three algorithms EQ (Execute Question), EG (Execute Goal) and ESG (Execute SubGoal).

The algorithm EQ is designed for modelling the procedural semantics of a goal clause. The algorithm EG (ESG) is intended to model the execution of a goal (subgoal), this means the successive attempts to make the goal (subgoal) hold.

Each algorithm is presented through the same frame : first, a description of the parameters and the remanent variables is provided, then the text of the algorithm is given.

But we begin by a quick outlook at the global variables.

2.4.1 Global variables :

These variables are known and are accessible (consulting and updating) by the algorithm EQ and all possible incarnations of any of the algorithms EG and ESG.

CX :

this variable contains the description of the current context and so, the logic program .

failure :

it is a boolean which is used for conveying the fact of success or failure of executions .

cut :

this boolean is used is used for conveying information about the encountered cuts.

2.4.2 Algorithm EQ :

(Execute Question or Execution Query)

The following algorithm provides the modelization of the procedural semantics of a query of the form

?- G .

where G is a list of subgoals.

Text :

```
EQ

variables     begin
                    V : VACV ;
                    E0 : stamp
              end

labels  begin
              next_CAS
        end

begin
        V := REFERENCE_ACV(← G) ;

        E0 := create( EG( G , V ) );

        next_CAS : execute( E0 ) ;

        if   failure    then print('no')
                        else begin
                                print( V ) ;
                                goto next_CAS
                             end
end.
```

2.4.3 Algorithm EG :

(Execute Goal)

Parameters :

This algorithm has two parameters. The first one must be a goal . The second one must be of VACV type. So, to create an incarnation of EG, the call to "create" must be of the form

create(EG(G , V))

where G is a goal and V a VACV .

Then, to execute it, one must use the stamp of the incarnation, returned by the creation, as parameter for a call to "execute". So, no more references to the parameters G and V are needed.

Text :

```
EG( G : goal ; var V : VACV )

remanent variables    begin
                          entry_pt : string / init := "prem" ;
                          Vrem : VACV / init := V ;
                          E1 : stamp ;
                          E2 : stamp
                      end

labels  begin
            prem ;
            next_empty ;
            next_SG_CAS ;
            next_CAS
        end
```

```
if G = () then
begin
        goto @ entry_pt ; { @ indicates indirection }

prem :  V := Vrem ;
        failure := false ;
        terminate(next_empty) ;

next_empty : failure := true ;
        cut := false ;
        selfkill
end
```

```
if G = (SG,G') then
begin
        goto @ entry_pt ;

prem :  V := Vrem ;
        E1 := create( ESG( SG , V ) ) ;

next_SG_CAS : execute( E1 ) ;
        if  failure  then  selfkill ;
        E2 := create( EG( G' , V ) ) ;

next_CAS : execute( E2 ) ;
        if  not failure  then   terminate( next_CAS ) ;
                        else    if  cut  then selfkill
                                        else goto next_SG_CAS
end
```

2.4.4 Algorithm ESG :

(Execute SubGoal)

Parameters and remanent variables :

ESG takes two parameters. The first one must be a subgoal . The second one must be of VACV type. So, to create an incarnation of ESG, the call to "create" must be of the form

create(ESG(SG , V))

where SG is a subgoal and V a VACV .

Then, to execute it, one must use the stamp of the incarnation, returned by the creation, as parameter for a call to "execute" .

ESG has five remanent variables : *Vrem, entry_pt, SG_instance, i* and *E1*. *E1* and *i* do not need to be initialized. *entry_pt* is initialized at creation to "prem" and *Vrem* to V. *SG_instance* must be initialized to the value of the instance of SG by the substitution defined by V !

Text :

```
ESG( SG : subgoal ; var V : VACV )

remanent variables    begin
                          Vrem : VACV / init := V ;
                          entry_pt : string / init := "prem" ;
                          SG_instance : subgoal / init := SG ΘV ;
                          E1 : stamp ;
                          i : integer
                      end

local variables begin
                      Vnew : VACV ;
                      θ : substitution ;
                      CL' : clause
                end
```

```
labels  begin
                prem ;
                next_clause ;
                next_CAS ;
                next_trial
        end
```

{
assume that

$SG\_instance = predicate(t_1 , \ldots , t_n)$

and that

$[\ CL_1 = predicate(s_{1\ 1} , \ldots , s_{1\ n}) :- G_1 ,$

$\ldots ,$

$CL_{nb\_cl} = predicate(s_{nb\_cl\ 1} , \ldots , s_{nb\_cl\ n}) :- G_{nb\_cl}\ ]$

is the list of clauses , of the program defined in the context CX, whose head is unifiable with *SG_instance*, provided that the variables appearing in them (the clauses) are renamed so they do not correspond to variables appearing in *SG_instance*
}

```
if SG_instance = "!" then
begin
        goto @ entry_pt ;

prem :  V := Vrem ;
        failure := false ;
        cut := true ;
        terminate( next_trial ) ;

next_trial : failure := true ;
        cut := true ;
        selfkill
end
```

```
if SG_instance ≠ "!" then
begin
        goto @ entry_pt ;

prem : i := 1 ;

next_clause : if  i > nb_cl  then      begin
                                              failure := true ;
                                              cut := false ;
                                              selfkill
                                       end ;
        CL' := STANDARDIZATION( CL_i , Vrem ) ;
                                { CL' = head(s'_i 1 , ... , s'_i n) :- G'_i }
        θ := MGU( CL' , SG_instance , Vrem ) ;
        Vnew := REFERENCE_ACV( CL' θ ) ;
        E1 := create( EG( G'_i θ , Vnew ) ) ;

next_CAS : execute( E1 ) ;
        if failure then  begin
                                if cut then begin
                                              cut := false ;
                                              selfkill
                                            end
                                else begin
                                        i := i + 1 ;
                                        goto next_clause
                                     end
                         end ;
        V := Vrem θ  ΘVnew;
        cut := false ;
        terminate( next_CAS )
end
```

2.5 Equivalence with the usual procedural semantics :

We present here the proposition that must hold in order to have the equivalence between our procedural semantics and the usual one founded on SLD-trees and the notion of computed answer substitution (CAS).

2.5.1 Sequence of answer substitutions :

To express the equivalence, we take over the concept of *sequence of answer substitution*s from [Deville 87]. Given an SLD-refutation procedure, a logic program P and a goal G, the sequence of answer substitutions for $P \cup \{\leftarrow G\}$ is the sequence of computed answer substitutions (CAS) for $P \cup \{\leftarrow G\}$, derived from the success branches eventually reached in the SLD-tree, according to the search rule. So, it is clear that corresponding to the sequence of CAS, we can also define the *sequence of success branches* and the *sequence of success nodes* for $P \cup \{\leftarrow G\}$.

We call *PROLOG-sequence of answer substitutions* the sequence of answer substitutions that would be computed by a PROLOG interpreter. The corresponding sequences of success branches and of success nodes are also called PROLOG-sequences.

In the rest of the work, when we speak of a sequence of answer substitutions, success branches or success nodes without indications about the SLD-refutation procedure, we assume it is the PROLOG-sequence.

Example :

consider the PROLOG program

```
p(_x, _z) :- q(_x, _y) , p(_y, _z) .
p(_x, _x) .
q(a, b) .
```

and the goal

```
p(_x,  b) .
```

The PROLOG-sequence of answer substitutions is the finite sequence θ_1,

θ_2 where

θ_1 = {_x/a}

θ_2 = {_x/b}

2.5.2 Proposition 2.3 (Equivalence) :

Proposition :

let P be a PROLOG program contained in CX
let G be a goal

if S is the PROLOG-sequence of answer substitutions for $P \cup \{\leftarrow G\}$,

then

- the execution of EQ prints a sequence of ACVs

  $V_1 , V_2 , V_3 , \ldots$

  such that ΘV_i is the i^{th} CAS of S.

- if S is finite, after the printing of the last V_i , the execution of EQ ends by printing "no" or endless continues (interpreter follows an infinite branch).

For EG and ESG, we prove in the next chapter some properties needed in order to justify the hereabove proposition. But, intuitively, we can give a rough (very rough) idea of what their results are.

2.5.3 EG :

if an incarnation E0 has been created with the call

create(EG (G , V))

then ,

execute(E0)

is such that when it terminates :

- the context *CX* can be changed
- if *failure* = false, V has been modified so it is an instance of its value as it was at creation of E0. This instance is such that G ΘV holds . The incarnation still lives.
- if *failure* = true and *cut* = true then, it is a cut encountered in G that is responsible of the failure . The incarnation is destroyed.
- - if *failure* = true and *cut* = false then, the failure is not due to a cut. The incarnation is destroyed.

2.5.4 ESG :

if an incarnation E0 has been created with the call

create(ESG (SG , V))

then ,

execute(E0)

is such that when it terminates :

- - the context *CX* can be changed
- - if *failure* = false, V has been modified so it is an instance of its value as it was at creation of E0. This instance is such that SG ΘV holds . The incarnation still lives.
- - if *failure* = true , then the incarnation is destroyed.
- - if *cut* = false , then SG ≠ "!".
- - if *cut* = true , then SG = "!".

CHAPTER 3 : EQUIVALENCE FOR FINITE SLD-TREES WITHOUT CUTS

3.1 INTRODUCTION :

The aim of this chapter is to provide the proof of the equivalence, between our procedural semantics and the usual one, when dealing with a program P and a goal clause G such that T(G) is finite and does not contain any cut ! These simplifications ease the proof and are lifted in further chapters.

Before proving that proposition 2.3 (Equivalence) is correct, we begin by the expression of properties of the algorithms ESG and EG. These properties are given by the following propositions and are the formal translations of the assertions we made over ESG and EG at the end of the previous chapter (see section 2.5) .

3.1.1 Proposition 3.1 :

Proposition :

let P be the program contained in CX
let SG be a subgoal (SG ≠ "!")
let V_0 be an ACV covering SG
let S = θ_1, ... , θ_m (m ≥ 0) be the sequence of answer substitutions for P ∪ {← SG ΘV_0}

if E0 is an incarnation of the algorithm ESG created with parameters SG and V = V_0 ,

then

- the j^{th} execution (1 ≤ j ≤ m) of E0 ends with
  V = V_0 θ_j ,
  failure = false ,

cut = false and
the incarnation E0 still alive .

- the $(m+1)^{th}$ execution of E0 ends with
  failure = true,
  cut = false and
  the incarnation E0 destroyed.

3.1.2 Proposition 3.2 :

Proposition :

let P be the program contained in CX
let G be a goal
let V_0 be an ACV covering G
let S = θ_1, ... , θ_m ($m \geq 0$) be the sequence of answer substitutions for $P \cup \{\leftarrow G \ \Theta V_0\}$

if E0 is an incarnation of the algorithm EG created with parameters G and $V = V_0$,

then

- the j^{th} execution ($1 \leq j \leq m$) of E0 ends with
  $V = V_0 \ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- the $(m+1)^{th}$ execution of E0 ends with
  failure = true,
  cut = false and
  the incarnation E0 destroyed.

3.1.3 Remarks

The proof for these last two propositions is made in three steps. The first step demonstrates that proposition 3.1 holds if proposition 3.2 holds. The second step

is just the contrary. The third step is needed to lift the vicious circle coming from the first two steps.

In the following sections, we speak of the first marginal execution, the second marginal execution, ... , the i^{th} marginal execution of an incarnation. The first marginal execution of an incarnation is the execution during which we start observation. The $(i+1)^{th}$ marginal execution of this incarnation ($i \geq 1$) is the first execution following its i^{th} marginal execution .

Our algorithms treat the goals as list of subgoals. It may happen that this list is empty () . In this situation, the SLD-tree for $P \cup \{\leftarrow ()\}$ is only composed of the root node $\leftarrow$ () . This root node can be seen as a success node because it is the empty clause. So, we can say that there is only one CAS, for $P \cup \{ \leftarrow ()\}$, which is the emtpy substitution .

3.2 PROOF OF PROPOSITION 3.1 IF PROPOSITION 3.2 IS CORRECT :

3.2.1 Preliminaries :

This section recalls some fundamental concepts about the depth-first search made by PROLOG in a SLD-tree. These are useful for the understanding of the proof. It also fixes some denominational conventions.

Let P be a program, SG be a subgoal and V_0 be an ACV. We consider the section of $T(SG\ \Theta V_0)$ concerning the i^{th} descendent input clause of $\leftarrow SG\Theta V_0$. We denote this clause CL'_i and its body G'_i . CL'_i is a variant of a program clause that we denote CL_i and whose body is G_i.

In $T(SG\ \Theta V_0)$, the root node is $\leftarrow SG\Theta V_0$ and it has a descendent $\leftarrow G'_i\gamma_i$ where γ_i is a MGU of the head of CL'_i and $SG\ \Theta V_0$. The tree is schematized in figure 3.1 .

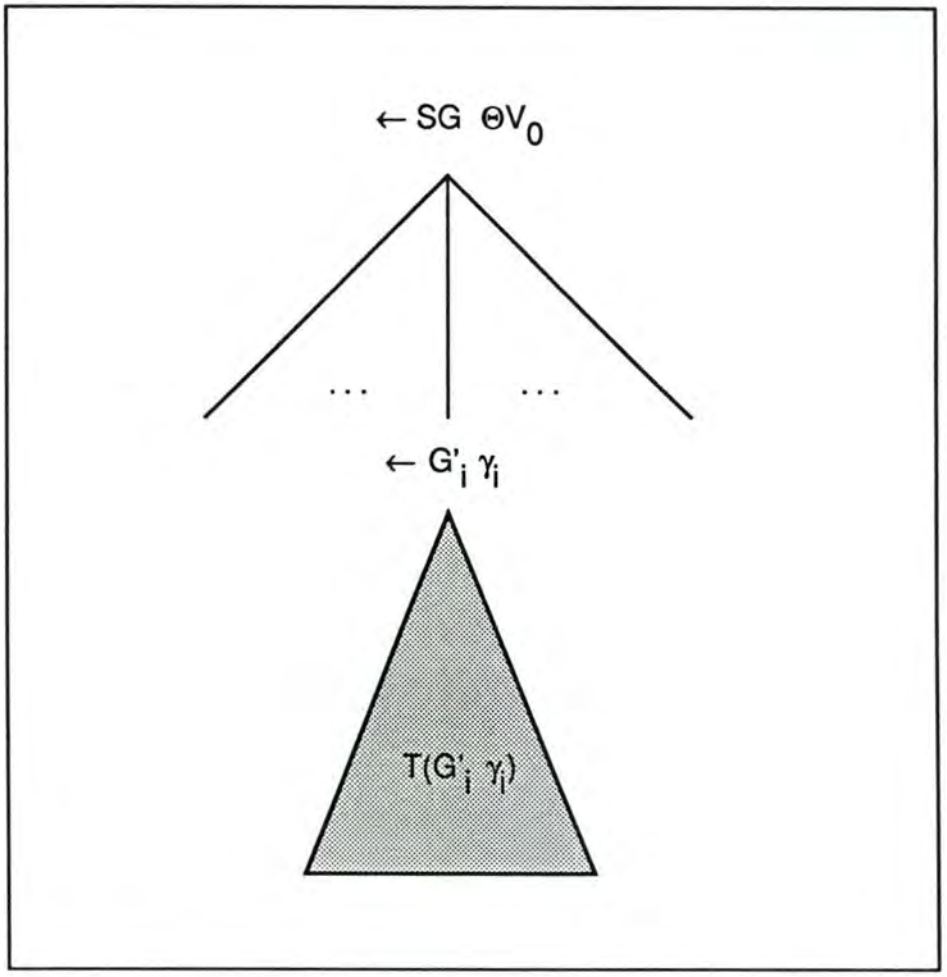

Figure 3.1

Now, we consider that T(SG ΘV_0) contains m success branches or, consequently, m CAS ($m \geq 0$) and that q ($0 \leq q \leq m$) branches of these success branches are passing by the node $\leftarrow G'_i \gamma_i$. If θ_1 , ... , θ_m is the sequence of answer substitutions for $P \cup \{\leftarrow SG\ \Theta V_0\}$ when using a depth-first search rule, this sequence is well the PROLOG-sequence of answer substitutions because there is no cut. Moreover, in this sequence, the q CAS whose corresponding success branches are passing by the node $\leftarrow G'_i \gamma_i$ are consecutive or form a subsequence. We can denote this subsequence $\theta_{(k+1)}$, ..., $\theta_{(k+q)}$ where $0 \leq k \leq m-q$, if the k first CAS are corresponding to success

branches located more at left than the ones passing by the node $\leftarrow G'_i \gamma_i$.

Given the definitions of SLD-trees and CAS, the properties of substitutions composition and the rule of depth-first search, we have that the sequence of answer subsitutions for $P \cup \{\leftarrow G'_i \gamma_i\}$ contains q CAS $\theta'_1, \ldots, \theta'_q$ and that $\theta_{(k+j)} = (\gamma_i \theta'_j)$ with $1 \leq j \leq q$.

3.2.2 Structure of the proof :

We provide here some guidelines through the proof by roughly explaining the contents of the lemmas and how they are used .

We consider an incarnation of the algorithm ESG created with SG and $V = V_0$. Figure 3.2 shows the global aspect of $T(SG \Theta V_0)$.

First, let us point that the variable i can be seen as the counter of descendent input clauses. Its value indicates which descendent input clause is under consideration. Remember that nb_cl indicates the number of descendent input clauses for the node $\leftarrow SG \Theta V_0$.

The aim of lemma 3.1 is to show that when an execution of the incarnation passes at the point labelled *next_clause* with $i >$ nb_cl, this execution must stop immediately with *failure* = false because there are no more possible input clauses .

Lemma 3.2 expresses that when an execution of the incarnation passes at the point labelled *next_clause* with $i \leq$ nb_cl , there will be a succession of executions of this incarnation before an execution comes back to that point but with i incremented by 1. The j^{th} execution of this succession terminates with $V = V_0 \theta_j$, θ_j being the j^{th} CAS, for $P \cup \{\leftarrow SG \Theta V_0\}$, such that the corresponding success branch is passing by the ith descendent node of $\leftarrow SG \Theta V_0$. The proof of this lemma uses the hypothesis that proposition 3.2 is correct , so that it is possible with an incarnation of EG to get the CAS of $T(G'_i \gamma_i)$.

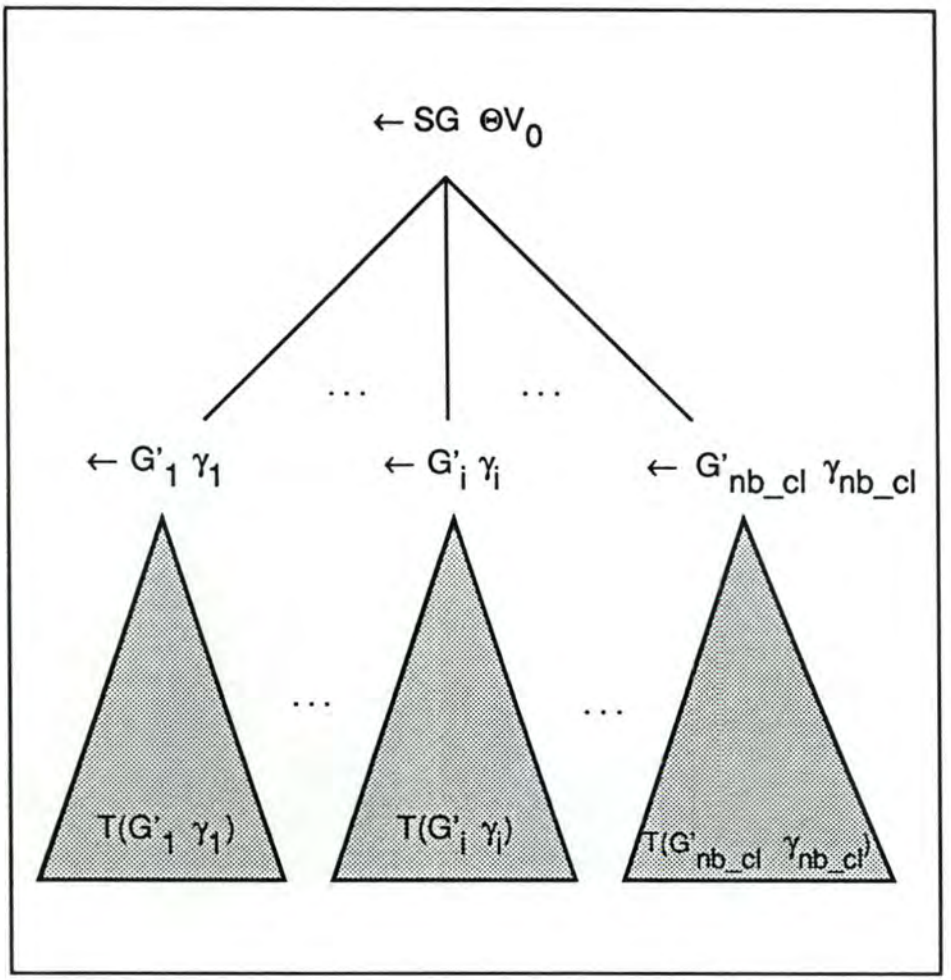

Figure 3.2

Then, we use lemma 3.2 in order to prove that given nc, an integer such that $1 \leq nc \leq nb\_cl$, there will be a succession of executions of the incarnation before an execution passes the point labelled *next_clause* with $i = nc + 1$. The j^{th} execution of this succession terminates with $V = V_0\ \theta_j$, θ_j being the j^{th} CAS, for $P \cup \{\leftarrow SG\ \Theta V_0\}$, such that the corresponding success branch is passing by one of the first nc^{th} descendent nodes of $\leftarrow SG\ \Theta V_0$. This is formalized in lemma 3.3.

The proof of proposition 3.1 uses lemma 3.3, for nc = nb_cl, together with lemma 3.1.

3.2.3 Lemma 3.1 :

Lemma :

let P be the program contained in CX
let SG be a subgoal
let V_0 be an ACV covering SG
let E0 be an incarnation of the algorithm ESG created with parameters SG and V = V_0

if a first marginal execution of E0 passes at the point labelled *next_clause* in the algorithm, with *i* = i1 and i1 > nb_cl

then, this first marginal execution ends with
failure = true ,
cut = false and
the incarnation E0 destroyed.

Proof :

it is immediate by symbolic execution of the "then" part of the alternative instruction labelled with *next_clause*.

qed

Note : this lemma holds, independently of proposition 3.2 !

3.2.4 Lemma 3.2 :

Lemma :

let P be the program contained in CX
let SG be a subgoal
let V_0 be an ACV covering SG
let E0 be an incarnation of the algorithm ESG created with parameters SG and V = V_0

if a first marginal execution of E0 passes at the point labelled *next_clause* in the algorithm, with *i* = i1 > 0 and i1 <= nb_ cl and

if $\theta_{(k+1)}$, ... , $\theta_{(k+q)}$ ($k \geq 0$ and $q \geq 0$) is the subsequence of answer substitutions for $P \cup \{\leftarrow SG\ \Theta V_0\}$ such that $\theta_{(k+j)}$ ($1 \leq j \leq q$) corresponds to a success branch passing by the node derived from the root of $T(SG\ \Theta V_0)$ and the $i1^{th}$ descendent input clause of $\leftarrow SG\ \Theta V_0$, then

- the j^{th} marginal execution of E0 (1 <= j <= q) ends with
  $V = V_0\ \theta_{(k+j)}$
  entry_pt = next_CAS
  failure = false
  cut = false and
  the incarnation E0 still alive .
- the $(q+1)^{th}$ marginal execution of E0 passes at the point labelled *next_clause* with *i* = i1 + 1 .

Proof :

We suppose that figure 3.1 (see section 3.2.1) provides the general form of $T(SG\ \Theta V_0)$.

As i1 <= nb_cl, the first marginal execution begins with the execution of the following instructions :

```
CL' := STANDARDIZATION( CL_i , Vrem ) ;
{1}
θ := MGU( CL' , SG_instance , Vrem ) ;
{2}
Vnew := REFERENCE_ACV( CL' θ ) ;
{3}
```

in {1} : from the specifications of STANDARDIZATION, we have CL' can be considered as the $i1^{th}$ input clause (CL'_{i1}) because it does not have any variable which already appears in $\leftarrow SG\ \Theta V_0$. This is also due to the fact that V_0 covers SG and Vrem = V_0 (because there is no instruction, in the algorithm ESG, that changes the value of Vrem).

in {2} : θ represents a MGU of the head of CL'_{i1} and $SG\,\Theta V_0$. So, $\theta = \gamma_{i1}$.

in {3} : Vnew contains a binding for all the variables appearing in $CL'_{i1}\,\gamma_{i1}$ and $\Theta Vnew = \varepsilon$. All the bindings of Vnew concern variables which do not appear in V_0. The value of Vnew here will be denoted $Vnew_0$.

Then, the instruction

E1 := create(EG($G'_i\,\theta$, Vnew)) ;

is executed and we know that $G'_i = G'_{i1}$ is the body of CL'_{i1} and that $\theta = \gamma_{i1}$.

Note that $G'_{i1}\,\gamma_{i1}\,\Theta Vnew = G'_{i1}\,\gamma_{i1}$ because $\Theta Vnew_0 = \varepsilon$.

By hypothesis, proposition 3.2 holds. This means that , as there are q CAS $\theta'_1, ..., \theta'_q$ (see section 3.2.1) for $P \cup \{\leftarrow G'_{i1}\,\gamma_{i1}\}$, we have

- the j^{th} execution ($1 \le j \le q$) of E1 ends with
  Vnew = $Vnew_0\ \theta'_j$,
  failure = false and
  the incarnation E1 still alive .
- the $(q+1)^{th}$ execution of E1 ends with
  failure = true ,
  cut = false and
  the incarnation E1 destroyed .

But, just after creation, the first execution of E1 occurs ! We distinguish two situations : q = 0 and q > 0 .

If q = 0 :

this first execution of E1 ends with
failure = true ,
cut = false and
the incarnation E1 destroyed

because $q < 1$.

So, the next executed instructions in E0 are

i := i + 1 ;
goto next-clause

Therefore, we can conclude that the first ($(q+1)^{th}$ with $q = 0$) marginal execution of E0 passes to the point labelled next_clause with $i = i1 + 1$.

If $q > 0$:

the first execution of E1 ends with

Vnew = $Vnew_0$ θ'_1 ,

failure = false and

the incarnation E1 still alive because $1 \leq q$.

So, the next executed instructions in E0 are :

V := Vrem θ ΘVnew ;
cut := false ;
terminate(next_CAS) ;

It is clear that after this sequence of instructions, we have that the execution of E0 is ended (E0 not destroyed) with *failure* = false , *cut* = false , *entry_pt* = next_CAS .

We must still prove that $V = V_0 \ \theta_{(k+1)}$.

But we have

- Vrem = V_0 (a)
- $\theta = \gamma_{i1}$ (b)
- Vnew = $Vnew_0 \ \theta'_1$ (c)
- $Vnew_0$ covers G'_{i1} because it covers CL'_{i1} (d)
- $\Theta Vnew_0 = \varepsilon$ (e)

- θ'_1 is a CAS for $T(G'_{i1}\ \gamma_{i1})(f)$

So, we can deduce that

$\Theta Vnew = \Theta(Vnew_0\ \theta'_1)$ (by c)

$= \langle\langle\ \Theta Vnew_0\ \theta'_1\ \rangle\rangle Vnew0$ (by proposition 2.2)

$= \langle\ \theta'_1\ \rangle Vnew_0$ (by e and proposition 2.1)

$= \theta'_1$ (by d, e and f)

and also that

$V = Vrem\ \theta\ \Theta Vnew$

$= V_0\ \gamma_{i1}\ \theta'_1$ (by a and b)

$= V_0\ \theta_{(k+1)}$ (see section 3.2.1)

Now, we have proved the thesis for j = 1 and, as *entry_pt* = next_CAS at the end of the first marginal execution of E0, we know that the next execution, the second (j = 2) marginal one, will begin at the point labelled *next_CAS*. Thus, it will begin by the second execution of the incarnation E1 of EG because the first marginal execution of E0 has performed one and only one execution of E1. So, by following an analoguous reasoning for this second execution of E0, then for the third one, and so on, we can prove the first part of the thesis for any value of j provided that $j \leq q$.

Note : this point should normally be proved using an inductive reasoning in order to be fully accurate. We think, however, that the hereabove explanations are clear enough to persuade oneself of the correctness of proposition 3.1 . If the reader is not at ease with the executions mechanisms of EG and ESG, we think it could be useful for him to build this recursive proof !

Finally, we must still prove that the $(q+1)^{th}$ marginal execution of E0 passes at the point labelled *next_clause* with $i = i1 + 1$.

We know that the q^{th} marginal execution has ended with *entry_pt* = next_CAS and that the next execution of E1 will be the $(q+1)^{th}$. So, the $(q+1)^{th}$ marginal execution of E0 begins by executing the $(q+1)^{th}$ execution of

E1 which ends, by hypothesis, with *failure* = true , *cut* = false and the incarnation E1 destroyed .

Then, as *failure* = true and *cut* = false, the following instructions are executed :

```
i := i + 1 ;
goto next_clause
```

It is clear that this $(q+1)^{th}$ marginal execution passes at *next_clause* with $i = i1 + 1$.

qed

3.2.5 Lemma 3.3 :

Lemma :

let P be the program contained in CX
let SG be a subgoal
let V_0 be an ACV covering SG

let nc be an integer such that $0 \leq nc \leq nb\_cl$
let E0 be an incarnation of the algorithm ESG created with parameters SG and $V = V_0$

if θ_1 , ... , θ_p ($0 \leq p$) is the subsequence of answer substitutions for $P \cup \{\leftarrow SG\ \Theta V_0\}$ such that θ_j ($1 \leq j \leq p$) corresponds to a success branch passing by one node $\leftarrow G'_k\ \gamma_k$ for some k such that $1 \leq k \leq nc$, G'_k being the body of the k^{th} descendent input clause of $\leftarrow SG\ \Theta V_0$ and γ_k a MGU of the head of this input clause and $SG\ \Theta V_0$, then

- the j^{th} ($1 \leq j \leq p$) execution of E0 ends with
  $V = V_0\ \theta_j$,
  failure = false ,

cut = false and
the incarnation E0 still alive .

- the $(p+1)^{th}$ execution of E0 passes at the point labelled *next_clause* with $i = nc + 1$.

Proof :

For this proof, we proceed by induction on nc .

Case 1 : $nc = 0$

This implies that the sequence of answer substitutions is empty ($p = 0$).

Due to the initialization of the remanent variable *entry_pt*, the first execution jumps to the point labelled *prem* where *i* is given the value 1. Then, it passes at the point labelled *next_clause* and we get the thesis.

Case 2 : $nc > 0$

By induction hypothesis, we know that if $\theta_1 , \ldots , \theta_q$ ($0 \le q \le p$) is the subsequence of answer substitutions for $P \cup \{\leftarrow SG\ \Theta V_0\}$ such that θ_j ($1 \le j \le q$) corresponds to a success branch passing by one of the nodes $\leftarrow G'_k\ \gamma_k$ ($1 \le k \le nc-1$) then

- the j^{th} ($1 \le j \le q$) execution of E0 ends with
  $V = V_0\ \theta_j$,
  failure = false ,
  cut = false and
  the incarnation E0 still alive .
- the $(q+1)^{th}$ execution of E0 passes at the point labelled *next_clause* with $i = nc$.

We can also say that the subsequence $\theta_{(q+1)} , \ldots , \theta_{(q+r)}$ ($0 \le r \le p-q$) is the subsequence of answer substitutions for $P \cup \{\leftarrow SG\ \Theta V_0\}$ such that $\theta_{(q+j)}$ ($1 \le j \le r$) corresponds to a success branch passing by the node derived from the root of $T(SG\ \Theta V_0)$ and the nc^{th} descendent input

clause of $\leftarrow$ SG ΘV_0 .

So, if we pick up the $(q+1)^{th}$ execution of EO when it passes at the point labelled *next_clause*, we can use lemma 3.2 with i = nc in order to deduce that

- the $(q+j)^{th}$ ($1 \leq j \leq r$) execution of E0 ends with
  $V = V0\ \theta_{(q+j)}$,
  failure = false ,
  cut = false and
  the incarnation E0 still alive .
- the $(q+r+1)^{th}$ or $(p+1)^{th}$ execution of EO passes at the point labelled *next_clause* with i = nc + 1 .

qed

3.2.6 Proof of proposition 3.1 :

It suffices to use lemma 3.3 with nc = nb_cl and, therefore, p = m to get immediately the first part of the thesis. The second one can be obtained by application of lemma 3.1 because, by lemma 3.3, we know that the $(m+1)^{th}$ execution of EO passes at the point labelled *next_clause* with i = nb_cl + 1 .

qed

3.3 PROOF OF PROPOSITION 3.2 IF PROPOSITION 3.1 IS CORRECT :

3.3.1 Preliminaries :

A. Stump :

Let T and T* be trees

T* is a *stump* of T iff

- T* is composed of a subset of nodes of T and a subset of branch of T .

- T and T* have the same root .
- if α is a node of T* , then the sequence of its descendents in T* is either exactly the same than in T or empty.

By sequence of descendents, we mean the set of descendents ordered in function of their position in the tree. We use the usual order going from left to right in the graphical representation.

Example :

Consider the trees of figure 3.3: T* is a stump of T but S* is not.

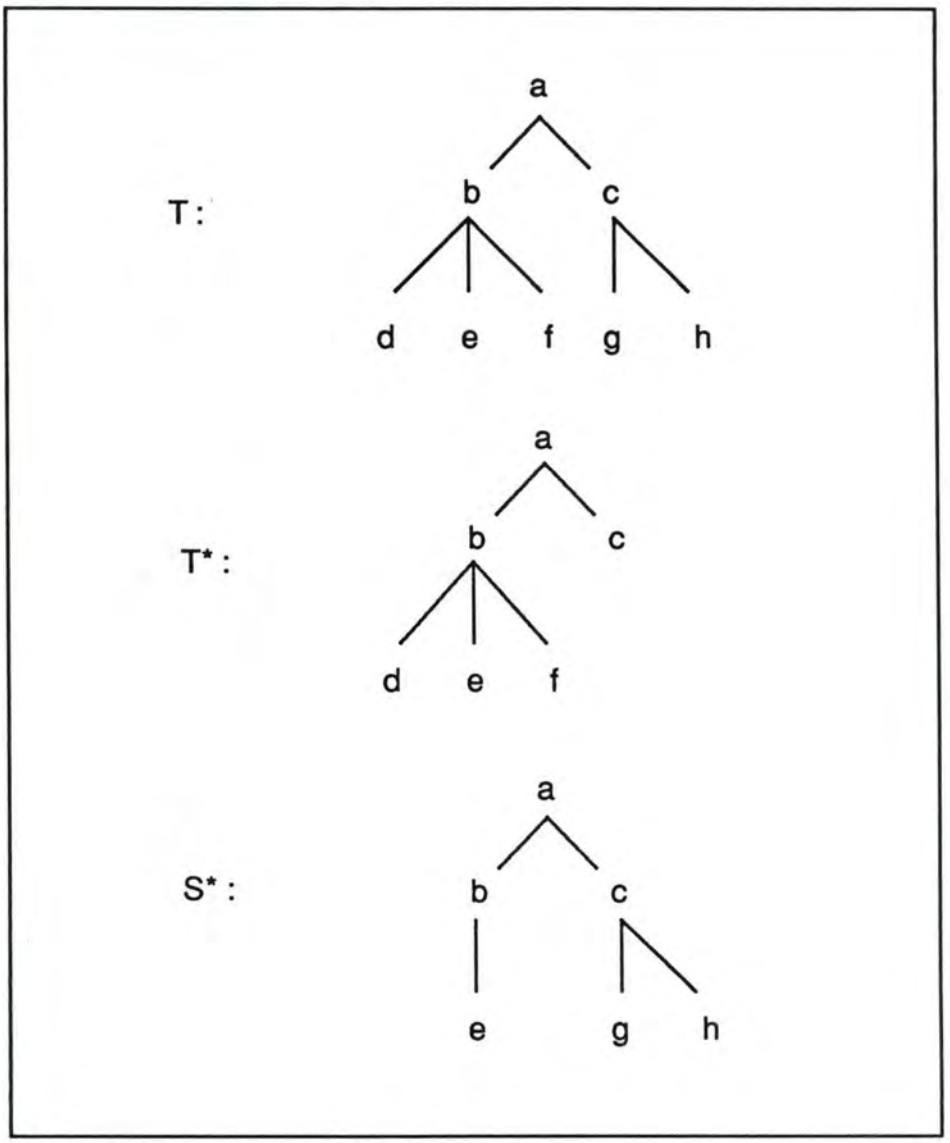

Figure 3.3

If we call terminal nodes the nodes which have no descendents, it is clear that the tree T can be obtained from the tree T* by grafting prolongations only to the terminal nodes of T* .

B. Using stump concept in SLD-trees :

Let P be a program, SG be a subgoal, G be a goal such that G = SG, G' .

Consider T*(G) a tree obtained from T(SG) simply by complying to the following rule of construction :

> if a node α in T(SG) has the form $\leftarrow G_\alpha$, the corresponding node in T*(G) has the form $\leftarrow G_\alpha$, (G' θ_α) , where θ_α is the composition of the unification substitutions used in the derivation from root to the node α in T(SG).

C. Proposition 3.3 :

Proposition :

> T*(G) is a stump of T(G) .

Proof :

> It is clear that T(G) and T*(G) have identical roots because G = SG, G' .
>
> We proceed then by contraposition :
>
> as the roots are identical, assuming that T*(G) is not a stump of T(G) means that there is a first level (the root composing level 1 and level i being the set of all the descendents of all the nodes composing level i-1) in T*(G), say level n (n > 1), where it is impossible to establish the equivalence between the sequence of descendents in T*(G) of a node α* of the previous level with the sequence of descendents of the same node in T(G).
>
> As level n is the first level where we encounter this impossiblility, it means that until level n-1, T*(G) corresponds to the definition of stump for T(G) .
>
> Now, we prove that it is impossible to have a such level.

The node α^* of the level n-1 has the form $\leftarrow G_\alpha$, $(G'\ \theta_\alpha)$ and has been obtained from a node α of T(SG) of the form $\leftarrow G_\alpha$.

The sequence of descendents of α^* in T*(G) is obtained from the sequence of descendents of α in T(SG). This implies that G_α is not the empty goal. Let $G_\alpha = SG_\alpha , G'_\alpha$.

In function of the SLD-derivation procedure of PROLOG, the sequence of descendents for α can be expressed as follows :

$$\leftarrow (G_1, G'_\alpha)\ \theta_1 , \ldots , \leftarrow (G_m, G'_\alpha)\ \theta_m$$

where m is the number of program clauses for which the head of a variant can be unified with SG_α . G_i is the body of the input clause corresponding to the i^{th} such program clause and θ_i is the MGU of the head of the input clause and SG_α.

So, the sequence of descendents for α^* has the following form, by definition of T*(G) :

$$\leftarrow (G_i , G'_\alpha)\ \theta_i , G'\ \theta_\alpha\ \theta_i \qquad 1 \le i \le m$$

or, equivalently,

$$\leftarrow (G_i , G'_\alpha , G'\ \theta_\alpha)\ \theta_i \qquad 1 \le i \le m \quad (I)$$

If we consider the node α^* in T(G), we know that it has the form

$$\leftarrow G_\alpha , G'\ \theta_\alpha$$

because α^* is at level n-1 and as we have seen that

$$G_\alpha = SG_\alpha , G'_\alpha$$

we can write

$$\leftarrow G_\alpha , G'\ \theta_\alpha = \leftarrow SG_\alpha , G'_\alpha , G'\ \theta_\alpha$$

As we consider the same program P than before and as the SLD-refutation procedure of PROLOG does not change, the sequence of descendents for α^* in T(G) can be expressed as follows :

$$\leftarrow (G_i , G'_\alpha , G' \theta_\alpha) \theta_i \qquad 1 \leq i \leq m \quad (II)$$

The sequences (I) and (II) are the same. So, it is impossible to have a first level for which the definition of stump is not respected. Therefore, we deduce that T*(G) is a stump of T(G).

qed

We call T*(G) the *first subgoal stump of T(G)*, T(SG) the *restricted first subgoal stump* and G' the *complement of T(SG) with respect to G*. When no ambiguity is possible, we simply talk about the *complement* of T(SG). The complement of a restricted first subgoal stump is the expression that must be added to the expression of the root in order to get the root of the first subgoal stump (not restricted). We also say that T(SG) is the restriction of T*(G).

We have already seen that it is possible to get T(G) from T*(G) by adding prolongations to the terminal nodes of T*(G).

Recall that terminal nodes for an SLD-tree are either success or failure nodes. The success (failure) nodes are the last nodes of successful (failed) derivations.

D. Proposition 3.4 :

Proposition :

It is possible to get T(G) from T*(G) by adding to each of its terminal node corresponding to a success node of T(SG) an appropriate prolongation. If α^* is a such terminal node , α the corresponding success node of T(SG) and θ_α the substitution defined by the success branch, of T(SG), ending in α , the prolongation to add to is T(G' θ_α).

Proof :

To prove proposition 3.4, we begin to prove that terminal nodes of T*(G) corresponding to failure nodes of T(SG) are also failure nodes for T(G).

Let α be a failure node, of T(SG), of the form

$$\leftarrow G_\alpha$$

As it is a failure node, we have G_α is not the empty goal and we can write that

$$G_\alpha = SG_\alpha , G'_\alpha$$

This node α has no descendents in T(SG) because it is impossible to find a clause in the program P for which the head of a variant can be unified with SG_α.

The node α^* of T*(G) corresponding to α has the form

$$\leftarrow G_\alpha , G' \theta_\alpha$$

or

$$\leftarrow SG_\alpha , G'_\alpha , G' \theta_\alpha$$

The corresponding node in T(G) has the same form; so, using the same SLD-refutation procedure than before, with the same program P, this node can only be a failure node.

Now, let α be a success node of T(SG). As a success node is an empty goal, the corresponding node in T*(G) has the form

$$\leftarrow G' \theta_\alpha$$

where θ_α is the CAS defined by the success branch, of T(SG), ending in α . Therefore, it is clear that the prolongation that must be attached to this node is T(G' θ_α) , the SLD-tree for P $\cup$ {$\leftarrow$ G' θ_α } .

qed

E. Consequences concerning the CAS of T(G) :

As a corollary of proposition 3.4, each success branch of T(G) must pass by a node corresponding to a success node of the restricted first subgoal stump.

So, as substitutions composition is associative, it comes immediately that the sequence of CAS defined by success branches passing by a node $\leftarrow G' \theta_\alpha$ can be obtained by composing θ_α with each CAS of the sequence of CAS for $P \cup \{\leftarrow G' \theta_\alpha\}$.

To get the sequence of answer substitutions for $P \cup \{\leftarrow G\}$ ($G = SG, G'$) , we can consider one by one the CAS for $P \cup \{\leftarrow SG\}$ (in the order defined by the sequence) and for each of them (denoted θ), take the sequence of their compositions with the CAS of the sequence of answer substitutions for $P \cup \{\leftarrow G' \theta\}$.

F. Graphical representation :

We propose here some conventions about the graphical representation of the subdivision of a SLD-tree into its first subgoal stump and the needed prolongations.

Assume that $G = SG , G'$. If $\theta_1, \ldots, \theta_q$ $(q \geq 0)$is the sequence of answer substitutions found in T(SG), we can sketch T(G) as in figure 3.4.

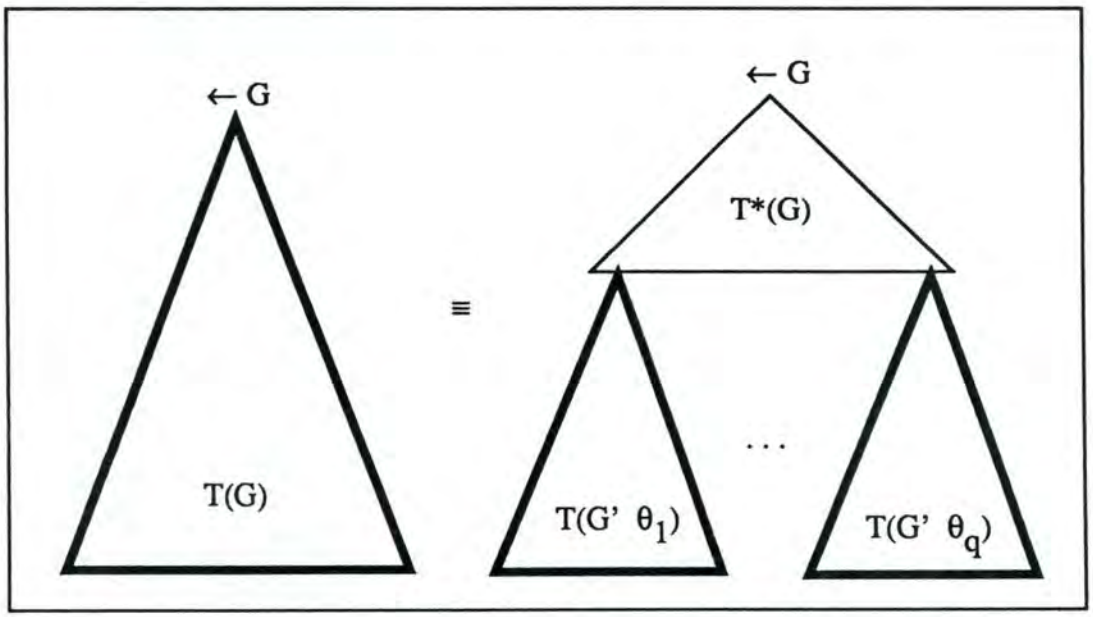

Figure 3.4

Example :

Consider the following program :

```
q(a,b).
s(b,c).
p(_x,_z) :- q(_x,_y), p(_y,_z) .
p(_x,_x).
r(_x,_y) :- q(_x,_y),s(_y,c) .
```

Figure 3.5 provides the sketch of T(G) for G = p(_x,b) , r(_x,b) .

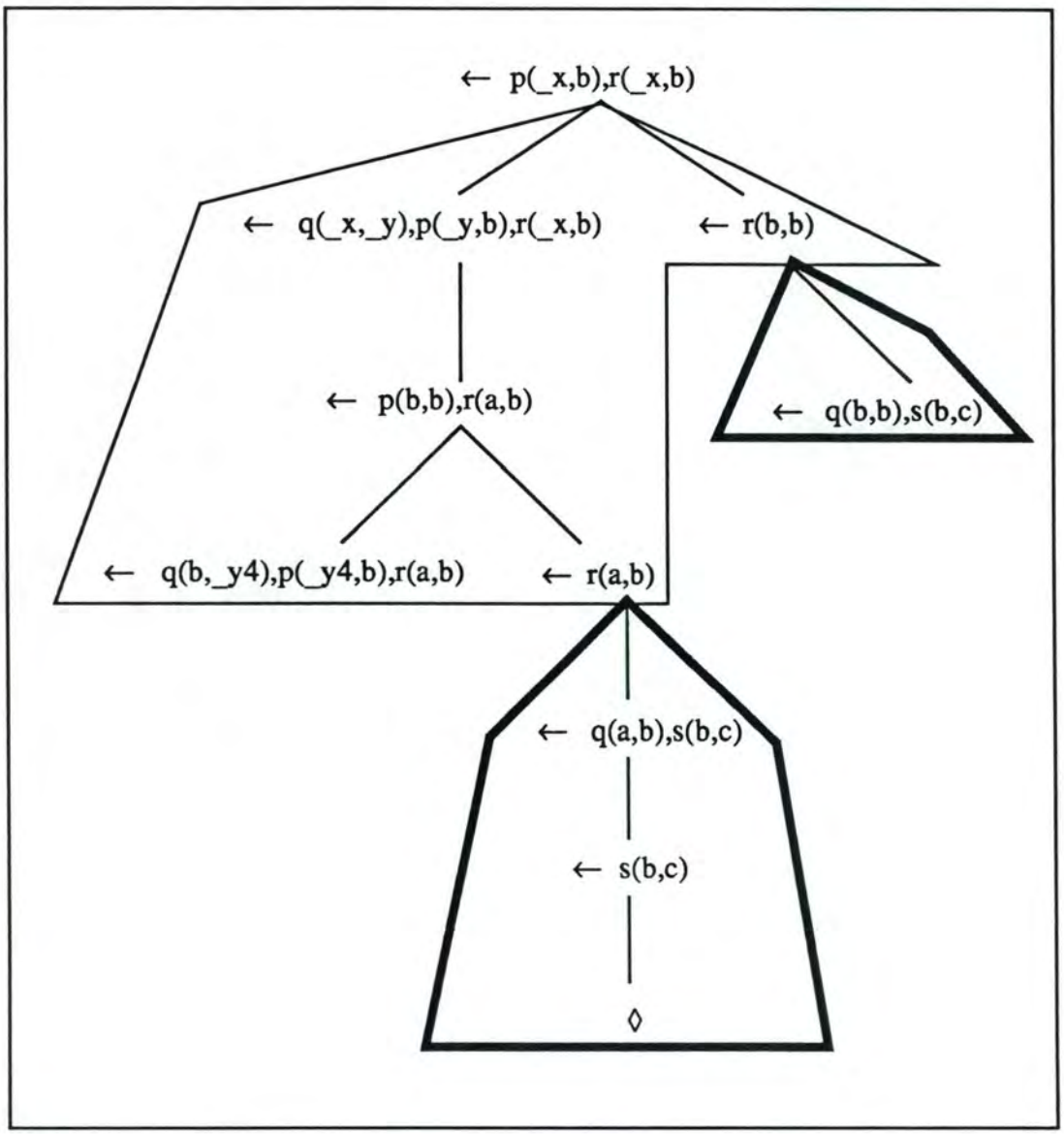

figure 3.5

3.3.2 Structure of the proof :

It is easy to proof the correctness of proposition 3.2 when G = () ! A quick glance at the algorithm is already persuasive. The case of G = () is used in our proof as the minimal case for an induction on the number of subgoals composing the goal.

The induction grounds on two lemmas ! We consider an incarnation of EG created with G and $V = V_0$. G = SG,G' is composed of n subgoals and we assume proposition 3.2 has been proved for goals of less than n subgoals. To understand the aim of these lemmas, the reader should better keep in mind the following illustration :

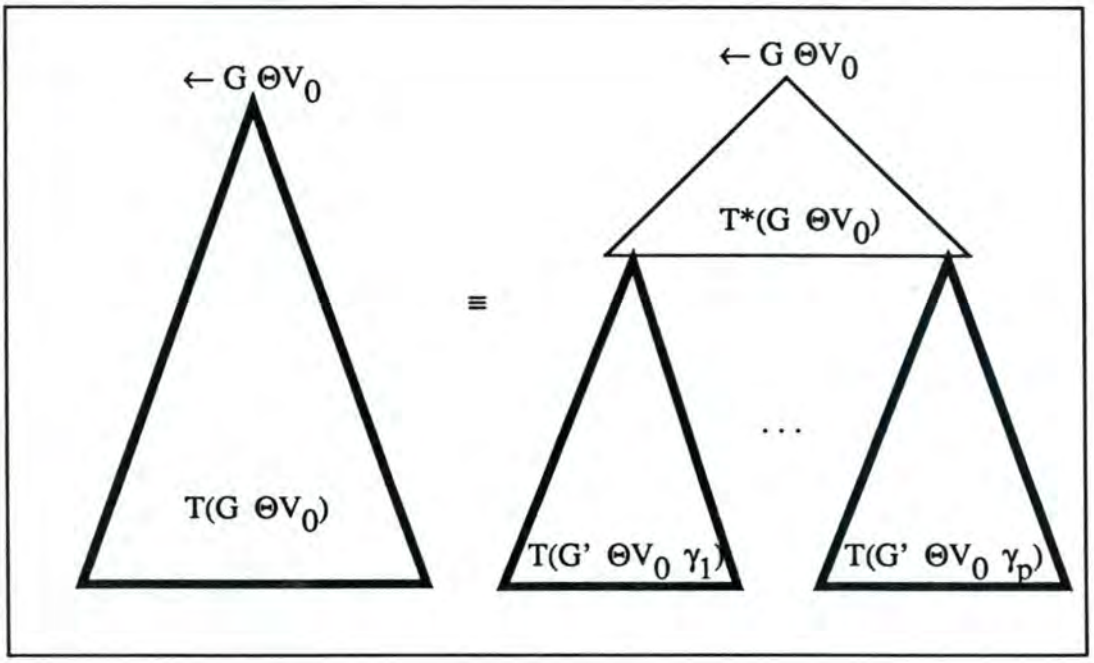

Figure 3.6

Lemma 3.4 expresses that if an execution of an incarnation E0 passes at the point where an incarnation E2 of EG is created and if $V = V_0 \theta$ then, there will be a succession of executions of the incarnation E0 before an execution passes at the point labelled *next_SG_CAS* . The j^{th} execution of this succession is so that it ends with $V = V_0 \theta \theta'_j$, θ'_j being the j^{th} CAS of the sequence of answer substitutions for $P \cup \{\leftarrow G' \Theta V_0 \theta\}$.

Then, we use this lemma and proposition 3.1 in order to prove that given i, an integer such that $1 \leq i \leq p$, there will be a succession of q executions ($q \geq 0$) of the

incarnation before an execution passes at the point labelled next_SG_CAS with the $(p+1)^{th}$ execution of the incarnation E1 to execute. The j^{th} execution of this succession terminates with $V = V_0\,\theta_j$, θ_j being the j^{th} CAS for $P \cup \{\leftarrow G\,\Theta V_0\}$.

Proposition 3.2 is easily deduced from this lemma and the characteristics (see proposition 3.1) of the $(p+1)^{th}$ execution of the incarnation E1 .

3.3.3 Lemma 3.4 :

We begin to recall the algorithm EG for non-empty goals and we add a label *L* that is used in the lemma :

```
if G = (SG,G') then
begin
        goto @ entry_pt ;

prem :  V := Vrem ;
        E1 := create( ESG( SG , V ) ) ;

next_SG_CAS : execute( E1 ) ;
        if  failure  then  selfkill ;

L :     E2 := create( EG( G' , V ) ) ;

next_CAS : execute( E2 ) ;
        if  not failure  then   terminate( next_CAS ) ;
                        else    if  cut  then selfkill
                                        else goto next_SG_CAS
end
```

Lemma :

let P be the program contained in CX
let G = SG, G' be a goal composed of n subgoals ($n \geq 1$)
let V_0 be an ACV covering G
let E0 be an incarnation of the algorithm EG created with parameters G and $V = V_0$

if algorithm EG respects proposition 3.2 for goals containing less than n subgoals and if a first marginal execution passes at the point labelled *L* with $V = V_1 = V_0\,\theta$ (θ being a substitution) and if θ_1, ..., θ_q ($q \geq 0$) is the sequence of answer substitutions for $P \cup \{\leftarrow G'\,\Theta V_1\}$, then

- the j^{th} ($1 \leq j \leq q$) marginal execution of E0 ends with
  $V = V_1\ \theta_j = V_0\ \theta\ \theta_j$,
  entry_pt = next_CAS,
  failure = false and
  the incarnation E0 still alive.
- the j^{th} ($1 \leq j \leq q$) marginal execution of E0 has performed the j^{th} execution of the incarnation E2 and has not performed any execution of the incarnation E1.
- the $(q+1)^{th}$ marginal execution of E0 passes at the point labelled *next_SG_CAS* and the incarnation E2 is destroyed.

Proof :

The execution begins with a creation of an incarnation of EG

E2 := create(EG(G' , V))

with $V = V_1$ (V_1 covers G' because it is an instance of V_0, V_0 which covers G).

By hypothesis, we can assert that

- the j^{th} ($1 \leq j \leq q$) execution of E2 ends with
  $V = V_1\ \theta_j = V_0\ \theta\ \theta_j$,
  failure = false and
  the incarnation E2 still alive.
- the $(q+1)^{th}$ execution of E2 ends with
  failure = true,
  cut = false and
  the incarnation E2 destroyed.

Then, we proceed by induction on the successive marginal executions of E0 .

Case 1 : first marginal execution

If q = 0 :

The first execution of E2, performed just after its creation, ends with *failure* = true , *cut* = false and the incarnation E2 destroyed .

So, the first marginal execution of E0 jumps to the point labelled *next_SG_CAS* and we get the thesis .

If q >= 1 :

The first execution of E2, performed just after its creation, ends with

$V = V_1\ \theta_1 = V_0\ \theta\ \theta_1$,

failure = false and

the incarnation E2 still alive ;

As *failure* = false, the execution of E0 executes

terminate(next_CAS)

so, it ends with

$V = V_1\ \theta_1 = V_0\ \theta\ \theta_1$,

failure = false ,

entry_pt = next_CAS and

the incarnation E0 still alive.

Moreover, E2 still lives and this first marginal execution of E0 has proceeded to the first execution of it . We also have that no execution of the incarnation E1 occurred.

Case 2 : j^{th} marginal execution if it is allright up to $(j-1)^{th}$ $(j-1 \leq q)$

By induction hypothesis, we have that the j^{th} marginal execution jumps at the point labelled *next_CAS* and performs the j^{th} execution of the incarnation E2 .

If $j = q + 1$:

We can assert that this jth execution of E2 ends with
failure = true ,
cut = false and
the incarnation E2 destroyed.

So, the jth marginal execution of E0 jumps to the point labelled *next_SG_CAS* and we get the thesis .

If $j <= q$:

The j^{th} execution of E2, performed just after its creation, ends with
$V = V_1\ \theta_j = V_0\ \theta\ \theta_j$,
failure = false and
the incarnation E2 still alive ;

As *failure* = false, the execution of E0 executes
terminate(next_CAS)
so, it ends with
$V = V_1\ \theta_j = V_0\ \theta\ \theta_j$,
failure = false ,
entry_pt = next_CAS and
the incarnation E0 still alive.

Moreover, E2 still lives and this first marginal execution of E0 has proceeded to the first execution of it . We also have that no execution of the incarnation E1 occurred.

qed

3.3.4 Lemma 3.5 :

Lemma :

let P be the program contained in CX
let G = SG , G' be a goal composed of n subgoals ($n \geq 1$)
let V_0 be an ACV covering G
let SN = s_1 ,..., s_p ($p \geq 0$) be the sequence of success nodes of

$T(SG\ \Theta V_0)$

let i be an integer such that $0 \le i \le p$
let E0 be an incarnation of algorithm EG created with parameters G and $V = V_0$

if algorithm EG respects proposition 3.2 for goals containing less than n subgoals and if $S = \theta_1, \ldots, \theta_q$ is the sequence of CAS for $P \cup \{\leftarrow G\ \Theta V_0\}$ such that, for each of them, the corresponding success branch passes by one of the nodes of the $T^*(G\ \Theta V_0)$ corresponding to one of the first i ($0 \le i \le p$) success nodes of SN, then

- the j^{th} ($1 \le j \le q$) execution of E0 ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- the $(q+1)^{th}$ execution of E0 passes at the point labelled *next_SG_CAS* and the next execution of the incarnation E1 will be the $(i+1)^{th}$ one .

Proof :

We proceed by induction on i.

Case 1 : i = 0

As *entry_pt* is initialized to *prem* and Vrem to V_0 at creation, the first execution jumps to the point labelled *prem* and executes

E1 := create(ESG(SG , V))

with $V = V_0$ (V_0 covers SG because it covers G).

Then, it passes at the point labelled *next_SG_CAS* and the next execution of E1 will be the first one. This provides the thesis because for i = 0, we have q = 0 .

Case 2 : $i \geq 1$ if it is allright up to $(i-1) < p$

The q CAS $\theta_1, \ldots, \theta_q$ can be subdivised as follows :

- $\theta_1, \ldots, \theta_{q1}$ ($0 \leq q1 \leq q$) the sequence of CAS for $P \cup \{\leftarrow G\ \Theta V_0\}$ such that, for each of them, the corresponding success branch passes by one of the nodes of the $T^*(G\ \Theta V_0)$ corresponding to one of the first (i-1) ($1 \leq i \leq p$) success nodes of $T(SG\ \Theta V_0)$.

- $\theta_{(q1+1)}, \ldots, \theta_{(q1+q2)}$ ($q1+q2 = q$) the sequence of CAS for $P \cup \{\leftarrow G\ \Theta V_0\}$ such that, for each of them, the corresponding success branch passes by the node of $T^*(G\ \Theta V_0)$ corresponding to the i^{th} success node of $T(SG\ \Theta V_0)$. (I)

By induction hypothesis, we get that

- the j^{th} ($1 \leq j \leq q1$) execution of E0 ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .

- the $(q1+1)^{th}$ execution passes at the point labelled *next_SG_CAS* and the last execution of the incarnation E1 will be the i^{th} one .

Now, at the point labelled *next_SG_CAS*, this $(q1+1)^{th}$ execution of E0 performs the i^{th} execution of E1 which ends with

$V = V_0\ \gamma_i$,
failure = false and
the incarnation E1 still alive ,

γ_i being the i^{th} CAS of the sequence of answer substitutions for $P \cup \{\leftarrow SG\ \Theta V_0\}$, by proposition 3.1 .

As *failure* = false , this $(q1+1)^{th}$ execution of E0 arrives at the point labelled *L* (see lemma 3.4) with $V = V_0\ \gamma_i$.

From (I) and from the developments we have made in section 3.3.1, we deduce that

$$\theta_{(q1+k)} = \gamma_i \ \theta'_k \ (1 \leq k \leq q2)$$

where θ'_k denotes the k^{th} CAS of $T(G' \ \Theta V_0 \ \gamma_i)$ which is the prolongation to add to the terminal node of $T^*(G \ \Theta V_0)$ corresponding to the i^{th} success node of $T(SG \ \Theta V_0)$.

We also have , at this point,

$G' \ \Theta V = G' \ «\Theta V_0 \ \gamma_i» V_0$ (by proposition 2.2)

$G' \ \Theta V = G' \ \Theta V_0 \ \gamma_i$

(by the the fact that γ_i is a CAS for $P \cup \{\leftarrow SG \ \Theta V_0\}$ and also the fact that V_0 covers G' because it covers G).

Now, we can use lemma 3.4 and also the fact that one more execution of E1 occurred to get the thesis.

qed

3.3.5 Proof of proposition 3.2 :

We proceed by induction on the number of subgoals composing the goal G .

Case 1 : 0 subgoal

It is immediate by using symbolic execution and the specifications of initialization of the remanent variables .

Case 2 : n (n > 0) subgoals and proposition 3.2 holds for goals of less than n subgoals.

Assume G = SG , G' .

Assume also that there are p CAS ($p \geq 0$) for $P \cup \{\leftarrow SG \ \Theta V_0\}$, we can use lemma 3.5 with i = p and q = m to get the first part of the thesis. The second part comes from the fact that the $(m+1)^{th}$ execution of E0 passes at

the point labelled *next_SG_CAS* and proceeds to the $(p+1)^{th}$ execution of E1 (see lemma 3.5). We know (proposition 3.1) that this execution of E1 ends with

failure = true ,
cut = false and
the incarnation E1 destroyed.

As *failure* = true , this $(m+1)^{th}$ execution of E0 performs selfkill.

qed

3.4 LIFTING CIRCULARITY :

Up to now, we have proved that proposition 3.1 is right if proposition 3.2 is so and vice versa. To lift the resulting circularity, we should give a simultaneous proof of propositions 3.1 and 3.2 . We do not give it here but we provide a detailed description of the way to follow to get this proof !

We assume the context CX contains the program P.

The depth of a SLD-tree is the length of its longest derivation (successful or failed) . If T is a SLD-tree, its depth is denoted depth(T).

What has to be shown is that $\forall n \in N$:

- $n = depth(T(SG\ \Theta V_0)) \Rightarrow$ proposition 3.1 holds for SG and $V = V_0$.
- $n = depth(T(G\ \Theta V_0)) \Rightarrow$ proposition 3.2 holds for G and $V = V_0$.

We use the induction principle on the depth of the trees.

We consider that the minimal cases concern SLD-trees of depth ≤ 1. These cases are discussed hereafter .

3.4.1 depth(T(SG ΘV_0)) = 0 :

The SLD-tree has the following form ::

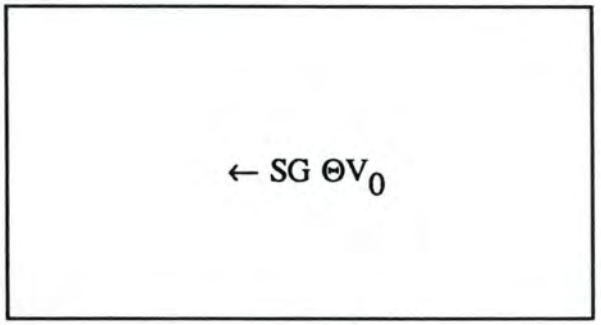

Figure 3.7

It is impossible to find an input clause, so nb_cl = 0 and there is no CAS !

The correctness of proposition 3.1 in this case can be proved by using lemma 3.1 which does not rely on the correctness of proposition 3.2 when nb_cl = 0 .

3.4.2 depth(T(G ΘV_0)) = 0 :

The SLD-tree has the form :

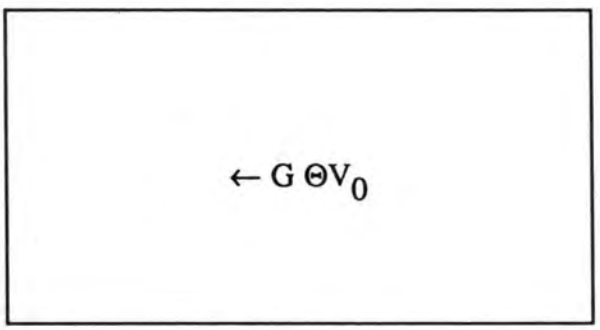

Figure 3.8

If G is not the empty goal, we can write that G = SG , G'. We also know that there is no CAS for P ∪ {← G ΘV_0}. So, if G = SG , G' , its restricted first subgoal stump is only composed of the root ← SG ΘV_0 .

The depth of this stump is also 0 and there is no CAS for $P \cup \{\leftarrow SG\ \Theta V_0\}$. By (I), we know that proposition 3.1 is respected for SG and $V = V_0$, so, we can take over lemma 3.5 to derive proposition 3.2 for SLD-trees of depth = 0 .

If G is the empty goal, the proof of proposition 3.2 remains the same.

3.4.3 depth(T(SG ΘV_0)) = 1 :

The SLD-tree has the following form

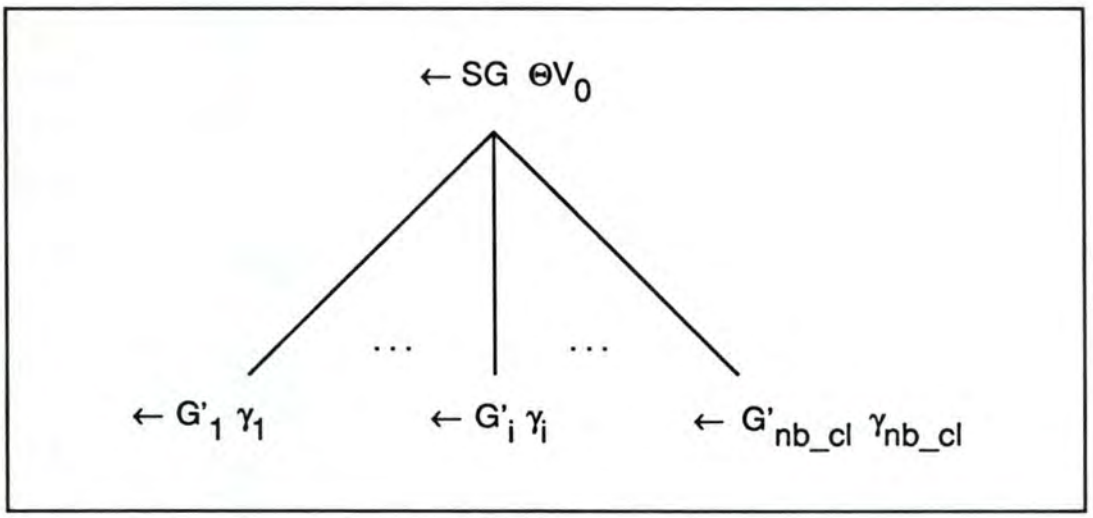

Figure 3.9

where G'_i is the body of the ith descendent input clause of $\leftarrow SG\ \Theta V_0$ and γ_i a MGU of its head and $SG\ \Theta V_0$.

Now, as depth($T(SG\ \Theta V_0)$) = 1 , we have depth($T(G'_i\ \gamma_i)$) = 0. Proposition 3.2 which holds for goals whose SLD-trees are of depth = 0 can be used.

3.4.4 depth(T(G ΘV_0)) = 1 :

As depth is 1, G is not the empty goal. Assume G = SG,G'.

The SLD-tree has the form :

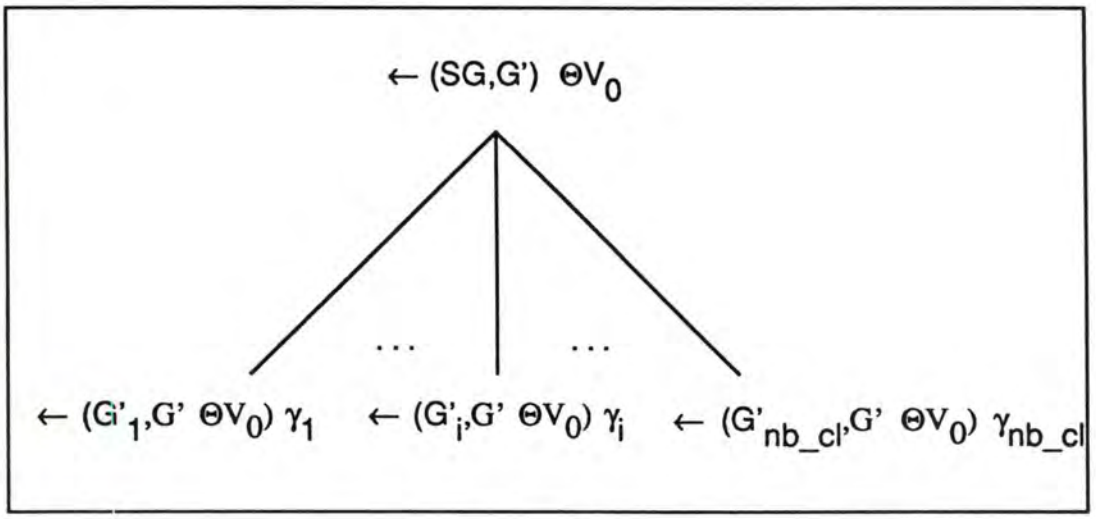

Figure 3.10

where G'_i is the body of the ith descendent input clause of $\leftarrow G\ \Theta V_0$ and γ_i a MGU of its head and $SG\ \Theta V_0$.

We can deduce that the tree is equal to its first subgoal stump whose restriction has the form presented at figure 3.9 .

The depth of this restriction is also 1 ! So, we have that proposition 3.1 holds for SG and $V = V_0$. It is then possible to prove that proposition 3.2 holds because when using, in the proof, the fact that proposition 3.1 must hold , it is enough to assume it holds for trees of depth ≤ 1 !

3.4.5 General case :

Now that we have shown how to cope with minimal cases, we consider the general case

Assuming that propositions 3.1 and 3.2 are correct for SLD-trees of depth = $n \leq n\text{-}1$, we have that

- $n = \text{depth}(T(SG\ \Theta V_0)) \Rightarrow$ proposition 3.1 holds for SG and $V = V_0$.

- $n = depth(T(G\ \Theta V_0)) \Rightarrow$ proposition 3.2 holds for G and $V = V_0$.

The correctness of proposition 3.1 for trees of depth n has to be established first ! This is due to the fact that it is enough to assume that proposition 3.2 holds for SLD-trees of depth (n-1) in this case . It is so because, when an incarnation of ESG uses an incarnation of EG, this latest concerns a subpart of $T(SG\ \Theta V_0)$ and the root of this subpart is a descendent of the root node of $T(SG\ \Theta V_0)$.

Now, if proposition 3.1 holds for SLD-trees of depth n , we can prove that proposition 3.2 also holds for such SLD-trees because when an incarnation of EG uses an incarnation of ESG, it is for the first subgoal of the goal. So, the incarnation of ESG is concerned with $T(SG\ \Theta V_0)$. But, this tree is the restriction of the first subgoal stump of $T(G\ \Theta V_0)$ and therefore, its depth is $\leq n$!

3.5 PROOF OF PROPOSITION 2.3 (Equivalence):

The beginning of the execution of EQ consists of the following instructions :

```
V := REFERENCE_ACV(← G) ;
{1}
E0 := create( EG( G , V ) ) ;
{2}
```

In {1} , we have that $V = V_0$ which covers G and $\Theta V_0 = \varepsilon$.

In {2} , we can deduce from proposition 3.2 that

- the j^{th} ($1 \leq j \leq m$) execution of E0 ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- the $(m+1)^{th}$ execution of E0 ends with
  failure = true ,
  cut = false and
  the incarnation E0 destroyed .

After creation, the first execution of E0 takes place.

If m = 0 :

this execution of E0 ends with *failure* = true , so, the execution of EQ prints "no" and ends immediately.

If m ≥ 1 :

this execution of E0 ends with V = $V_0\ \theta_1$, *failure* = false and the incarnation E0 still alive . So, the execution of EQ continues by printing V. But we have, when V is printed, that

- $\Theta V_0 = \varepsilon$. (a)
- V_0 covers G . (b)
- θ_1 is a CAS for $P \cup \{\leftarrow G\ \Theta V_0\}$ and thus a substitution for variables appearing in G . (c)

From these assertions, we can deduce that

$$\begin{aligned} \Theta V &= \langle\langle \Theta V_0\ \theta_1 \rangle\rangle V_0 && \text{(by proposition 2.2)} \\ &= \langle \theta_1 \rangle V_0 && \text{(by a and proposition 2.1)} \\ &= \theta_1 && \text{(by a, b and c)} \end{aligned}$$

After the printing , the execution of EQ jumps to the point labelled *next_CAS*.

Now, assuming that proposition 2.3 is correct up to the $(j-1)^{th}$ CAS of S ($1 < j < m$) and that after printing the $(j-1)^{th}$ ACV, the execution of EQ jumps at the point labelled *next_CAS* and also that the last execution of E0 was the $(j-1)^{th}$, we can prove that proposition 2.3 is also correct up to the j^{th} CAS of S.

The execution of EQ passing at the point labelled *next_CAS* performs the j^{th} execution of E0 .

If $j = m+1$:

this execution of E0 ends with *failure* = true , so, the execution of EQ prints "no" and ends immediately.

If $j \leq m$:

this execution of E0 ends with $V = V_0\ \theta_j$, *failure* = false and the incarnation E0 still alive . So, the execution of EQ continues by printing V. But we have, when V is printed, that

- $\Theta V_0 = \varepsilon$. (a)
- V_0 covers G . (b)
- θ_j is a CAS for $P \cup \{\leftarrow G\ \Theta V_0\}$ and thus a substitution for variables appearing in G . (c)

From these assertions, we can deduce that

$$\begin{aligned} \Theta V &= \langle\langle \Theta V_0\ \theta_j \rangle\rangle V_0 && \text{(by proposition 2.2)} \\ &= \langle \theta_j \rangle V_0 && \text{(by a and proposition 2.1)} \\ &= \theta_j && \text{(by a, b and c)} \end{aligned}$$

After the printing , the execution of EQ jumps to the point labelled *next_CAS*.

The last execution of E0 was also the j^{th} one.

qed

So, we have proved the equivalence of our expression of the procedural semantics and the usual one based on SLD-trees when dealing with finite SLD-trees where no cut appears.

CHAPTER 4 : EQUIVALENCE FOR INFINITE SLD-TREES WITHOUT CUTS

4.1 INTRODUCTION :

In chapter 3, the proof of the equivalence uses the induction principle on the depth of SLD-trees. Therefore, this proof cannot hold for infinite SLD-trees. In this part of the work, we show how to deal with them.

In this case, the PROLOG-sequence of CAS, denoted S in proposition 2.3 (equivalence), contains all the CAS whose success branches are located more at left in the SLD-tree than the leftmost infinite branch. It is clear that this sequence can be finite or infinite ! If it is finite, it means that the search, after having found the last CAS, goes desperately seeking for another one along the infinite branch.

The proof of proposition 2.3 for finite SLD-trees shows that the algorithm EQ relies completely on the algorithm EG. In the case of infinite SLD-trees, we can take over the same reasoning, provided we can assert the following proposition (derived from proposition 3.2) about the algorithm EG .

4.1.1 Proposition 4.1 :

Proposition :

let P be the program contained in CX
let G be a goal
let V_0 be an ACV covering G

let S be the sequence of answer substitutions for $P \cup \{\leftarrow G\ \Theta V_0\}$

let $T(G\ \Theta V_0)$ contain at least one infinite branch

let E0 be an incarnation of the algorithm EG created with parameters G and $V = V_0$

if S is a finite sequence of m CAS, $\theta_1, ..., \theta_m$, then

- the j^{th} execution of E0 ($1 \leq i \leq m$) ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- the $(m+1)^{th}$ execution of E0 never ends .

if S is an infinite sequence of CAS, $\theta_1, \theta_2, \theta_3, ...$, then

- the j^{th} execution of E0 ($j > 0$) ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .

To prove this proposition, we need some new concepts that are presented in the next section. Then, two propositions about these concepts are presented before we focus on the proof for SLD-trees containing one and only one infinite branch because it is far more easier to understand than the proof for any infinite SLD-tree ! Finally, we show how to deal with SLD-trees containing more than one infinite branch.

4.2 SUBGOAL SUBTREES (SS), SUBGOAL RESTRICTED SUBTREES (SRS) :

In this section, we assume that

- P is a program.
- $G = SG_1 , ... , SG_n$ ($n \geq 1$) is a goal of n subgoals.

We divide T(G) into specific subparts which receive the generic denomination of *Subgoal Subtrees* (SS). These subparts are composed of a subset of the nodes of T(G) and a subset of its branches. All the subparts we consider have the following characteristics :

- they are trees .
- if a subpart contains a node α which belongs to T(G) (where he has descendents) then it contains either all the descendents of α appearing in T(G), or none of them.

We can immediately see the similarity with the concept of stump (see section 3.3.1). This is normal, because these subdivisions of an SLD-tree are defined by using this concept.

We distinguish the *SS of the first degree* (SS1), *of the second degree* (SS2), ...

We begin to define the SS1.

4.2.1 SS1 :

If G is composed of n subgoals, there can be n classes, n levels of SS1x !

At the first level, there is only one *First Subgoal Subtree of degree 1* for T(G), 1SS1(G), it is in fact the first subgoal stump of T(G) : T*(G). Corresponding to this 1SS1, we have a *First Subgoal Restricted Subtree of degree 1* for T(G), 1SRS1(G), which is the restriction of the first subgoal stump.

The complement C of the 1SRS1(G) is the expression to add to its root in order to get the root of the corresponding 1SS1(G). Here, we have $C = SG_2 , \ldots , SG_n$. *To complete* a 1SRS1(G) is to perform the treatment that permits to get the 1SS1(G) from the 1SRS1(G) and its complement (see section 3.3.1).

From section 3.3.1 , we know that we can get T(G) simply by adding prolongations to the nodes of its 1SS1 corresponding to success nodes of the 1SRS1(G). These terminal nodes of the 1SS1 are called the *grafting nodes* of the 1SS1(G).

Each prolongation has a root of the form

$$\leftarrow C\,\theta$$

or

$$\leftarrow (SG_2 , \ldots , SG_n)\,\theta$$

where θ is the CAS defined by the success branch of the 1SRS1(G) ending at the success node corresponding to the grafting node, of the 1SS1(G), on which the

prolongation is fastened . This substitution is called a *First Subgoal Grafting Substitution of degree 1* for T(G), 1SGS1(G). Each prolongation is called a *First Subgoal Prolongation of degree 1* for T(G), 1SP1(G) .

Now, we can recursively define the i^{th} level : *i^{th} Subgoal Subtrees of degree 1* for T(G) (iSS1(G)), *i^{th} Subgoal Restricted Subtrees of degree 1* for T(G) (iSRS1(G)), *i^{th} Subgoal Grafting Substitutions of degree 1* for T(G) (iSGS1(G)) and *i^{th} subgoal Prolongations of degree 1* for T(G) (iSP1(G)) with $2 \leq i \leq n$.

An iSS1(G) is the first subgoal stump of an (i-1)SP1(G) and the corresponding iSRS1(G) is the restriction of this first subgoal stump. So, an iSRS1(G) is a SLD-tree for $P \cup \{\leftarrow SG_i\ \sigma\}$ where σ is the (i-1)SGS1(G) corresponding to the grafting node on which the (i-1)SP1(G) is fastened. The complement of an iSRS1(G) has the form

$$(SG_{(i+1)}, \ldots, SG_n)\ \sigma$$

Equivalently, we have that an iSS1(G) is the $1SS1((SG_i,\ldots,SG_n)\ \sigma)$ of an (i-1)SP1(G) and an iSRS1(G) is the $1SRS1((SG_i,\ldots,SG_n)\ \sigma)$ of an (i-1)SP1(G).

An iSP1(G) is a $1SP1((SG_i,\ldots,SG_n)\ \sigma)$ of an (i-1)SP1(G). Each iSP1(G) is a SLD-tree for $P \cup \{\leftarrow (SG_{(i+1)}, \ldots, SG_n)\ \sigma\ \gamma\}$ where σ is the (i-1)SGS1(G) corresponding to the grafting node the (i-1)SP1(G) , containing the iSP1(G), is fastened on and γ is the CAS defined by the success node of the iSRS1(G) corresponding to the grafting node, of the iSS1(G), the iSP1(G) is fastened on. $\sigma\gamma$ is an iSGS1(G) ; it is in fact the composition of the unification substitutions defined along the branch going from the root of T(G) to a grafting point of the iSS1(G) .

Note that , by convention, we say T(G) is the 0SP1(G). When no ambiguity is possible, we drop the subscripts and simply speak of iSS1 , iSRS1, iSGS1 and iSP1 !

4.2.2 Graphical representation :

The graphical representation of the concepts we present in the previous subsection is an "inductive" application of what we explain in section 3.3.1 .

Figure 4.1 provides the sketches for T(G) and its subdivision in 1SS1, 1SP1 ,2SS1 and 2SP1. It is easy to imagine the following subdivisions into higher levels SS1 and SP1, .

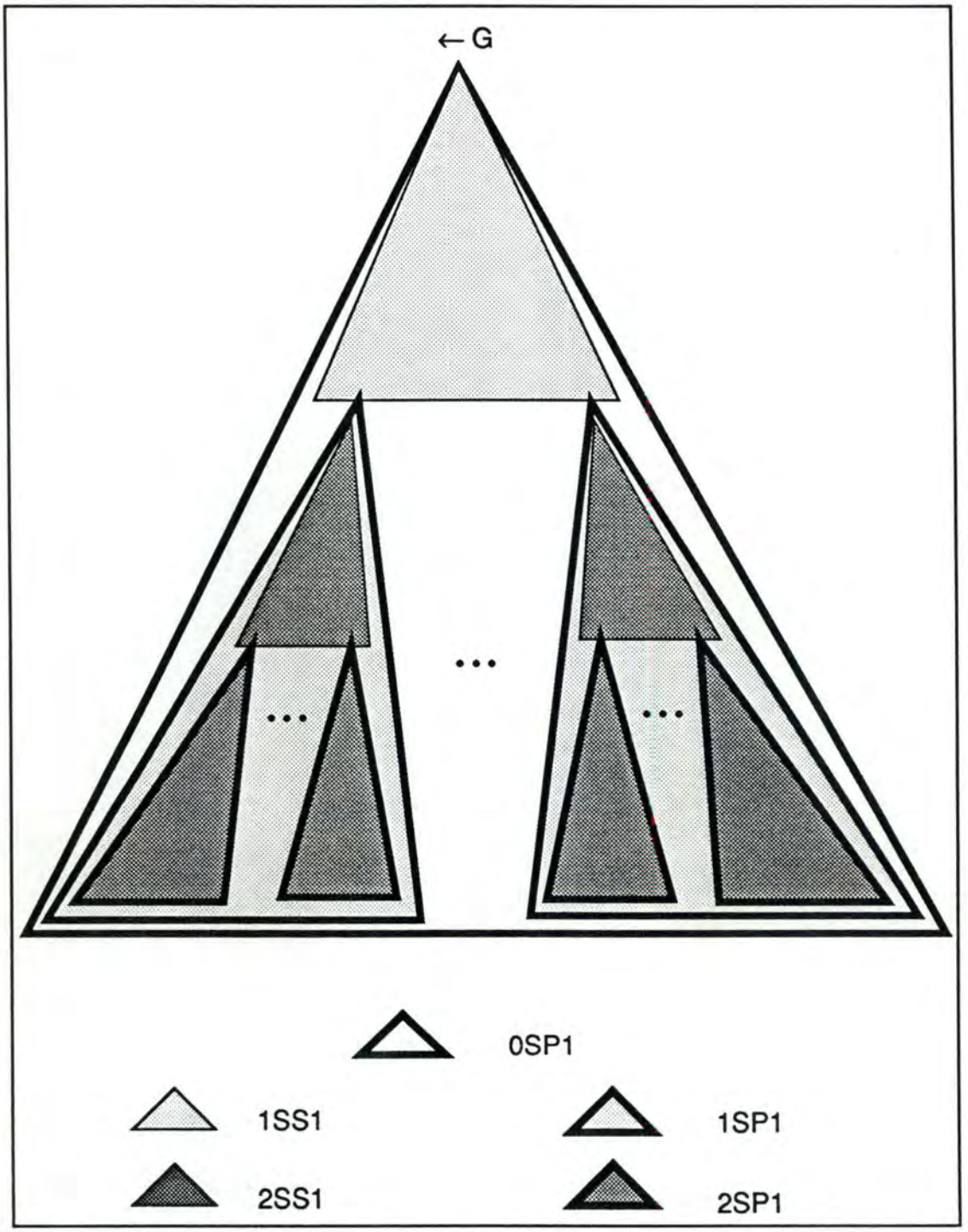

Figure 4.1

Example :

Consider the goal clause (p(_x,b) , r(_x,b) , t(_x,b)) with the following program

```
p(_x,_z) :- q(_x,_y) , q(_y,_z) .
p(_x,_x) . q(a,b) .
r(_x,_y) :- s(_x,_y) .
s(a,b) .
t(a,b) .
```

Here is the resulting SLD-tree where 1SS1, 1SP1, 2SS1 and 2SP1 are shown :

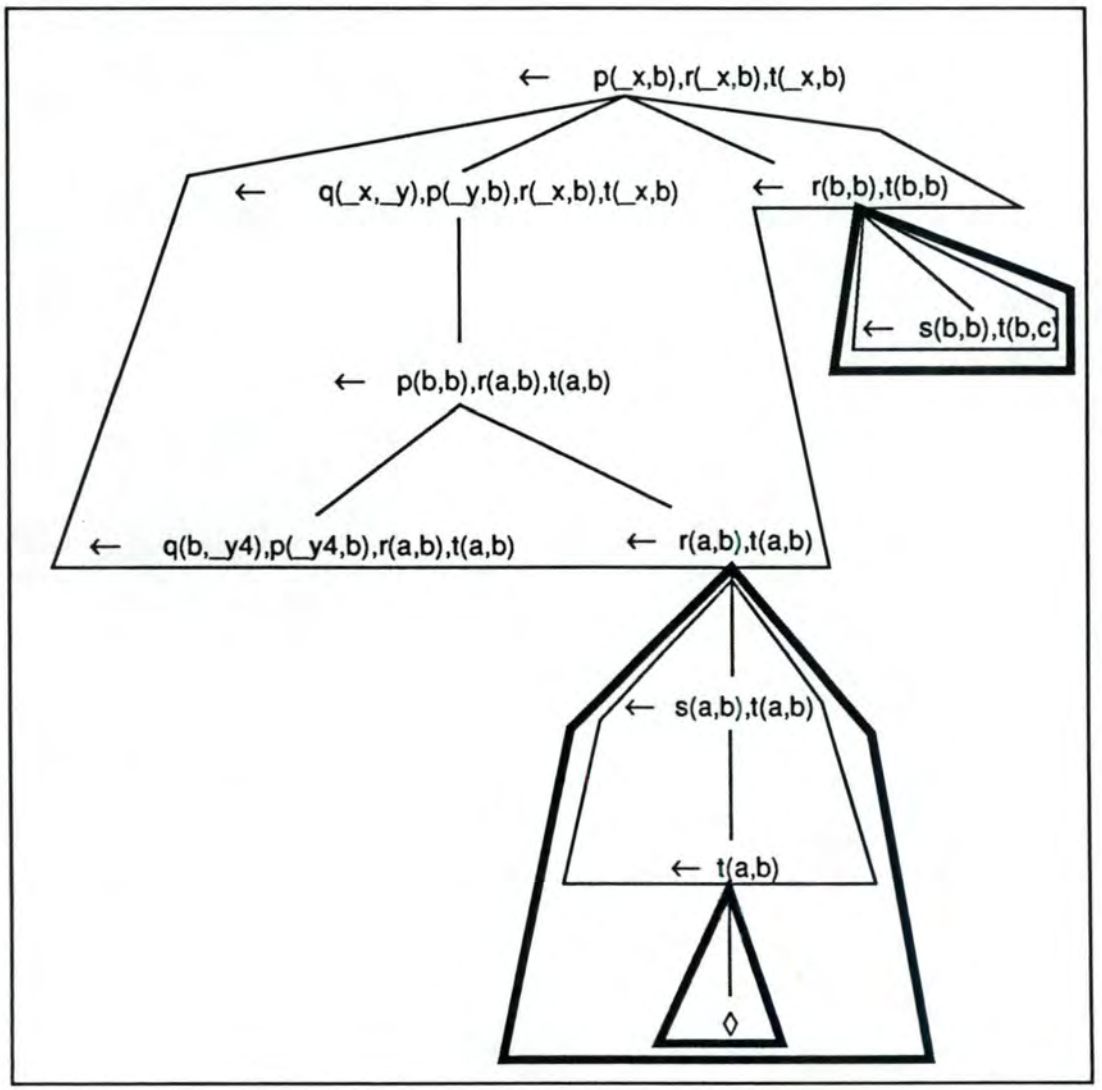

Figure 4.2

But we need some more conventions for reprseentation of SS and SP with infinite branches! These are illustrated in figure 4.3 .

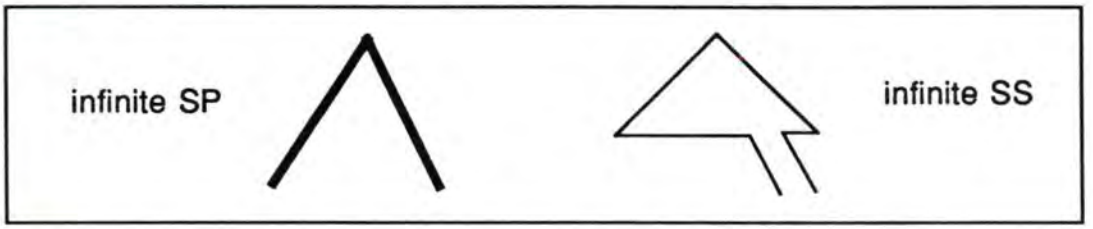

Figure 4.3

Note however that we do not represent the SS which are composed only of their root node !

4.2.3 SSj :

Now, we turn to the definition of the j^{th} degree SS and SRS. Formally, this should be recursive, defining j^{th} degree SS and SRS from the $(j-1)^{th}$ degree SRS. However, for simplicity , we do not provide here a formally accurate and general definition ; in fact, it is enough to have a good idea of the mechanism and we think a way to achieve this is to show how to define the second degree SS and SRS, starting from SRS1.

Note : we do not define prolongations and grafting substitutions of a degree superior to 1 !

If we consider an iSRS1, it has the form given at the following figure :

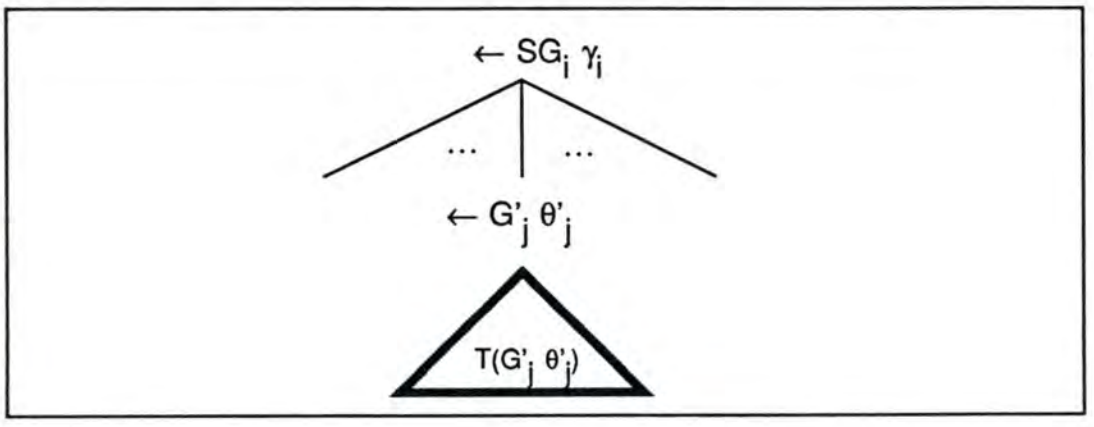

Figure 4.4

Where G'_j is the body of the jth descendent input clause and θ'_j a MGU of the head of this input clause and $SG_i\ \gamma_i$.

Let $G'_j = SG'_1, \ldots, SG'_p$.

$T(G'_j\ \theta'_j)$ can be subdivised into first degree subparts. The SRS1 for $T(G'_j\ \theta'_j)$ are SRS2 for T(G). But a SS1 for $T(G'_j\ \theta'_j)$ is **NOT** a SS2 for T(G).

Consider a kSRS1 for $T(G'_j\ \theta'_j)$, it is the SLD-tree for $P \cup \{\leftarrow SG'_k\ \gamma'_k\}$ where γ'_k is the (k-1)SGS1 for $T(G'_j\ \theta'_j)$ associated to the grafting node which corresponds to the root node of the kSS1 for T(G'j θ'j) whose restriction is the kSRS1 at hand.

We know that the complement for the iSRS1 for T(G) has the form

$$(SG_{(i+1)}, \ldots, SG_n)\ \gamma_i = C$$

and that the complement of the kSRS1 for T(G'j 'j) has the form

$$(SG'_{(k+1)}, \ldots, SG'_p)\ \gamma'_k = C'$$

If we complete with C' the kSRS1 for $T(G'_j\ \theta'_j)$, we get the corresponding kSS1 for $T(G'_j\ \theta'_j)$ which is a subtree of $T(G'_j\ \theta'_j)$ and therefore a subtree of the iSRS1 for T(G). From section 3.3.1 , we know that when we complete this iSRS1 with C, the expression that we add to the expression of the node $\leftarrow G'_j\ \theta'_j$ is $C\ \theta'_j$ and the one to the expression of the grafting node corresponding to the root of the kSS1 for $T(G'_j\ \theta'_j)$ is $C\ \theta'_j\ \gamma'_k$.

So, if we complete the kSRS1 for $T(G'_j\ \theta'_j)$, which is a kSRS2 for T(G), with the expression $(C',\ C\ \theta'_j\ \gamma'_k)$, we get a subtree of T(G) and it is this subtree which is a kSS2 for T(G) ! The expression $(C',\ C\ \theta'_j\ \gamma'_k)$ is called the *second degree complement* of the kSRS2 while C' is the *first degree complement.*

Using the same reasoning , it is possible to define SSj and SRSj from SRS1 of SRS(j-1) . To get an SSj from its corresponding SRSj, we must complete this

last one with a j^{th} degree complement which can be obtained from the first degree complement of SRSj and a proper instance of the $(j-1)^{th}$ degree complement of the SRS(j-1) .

Example :

We can take over the example already treated in figure 4.2 . Figure 4.5 shows one of the 1SS1 subdivised into 1SS2 and 1SP2. .

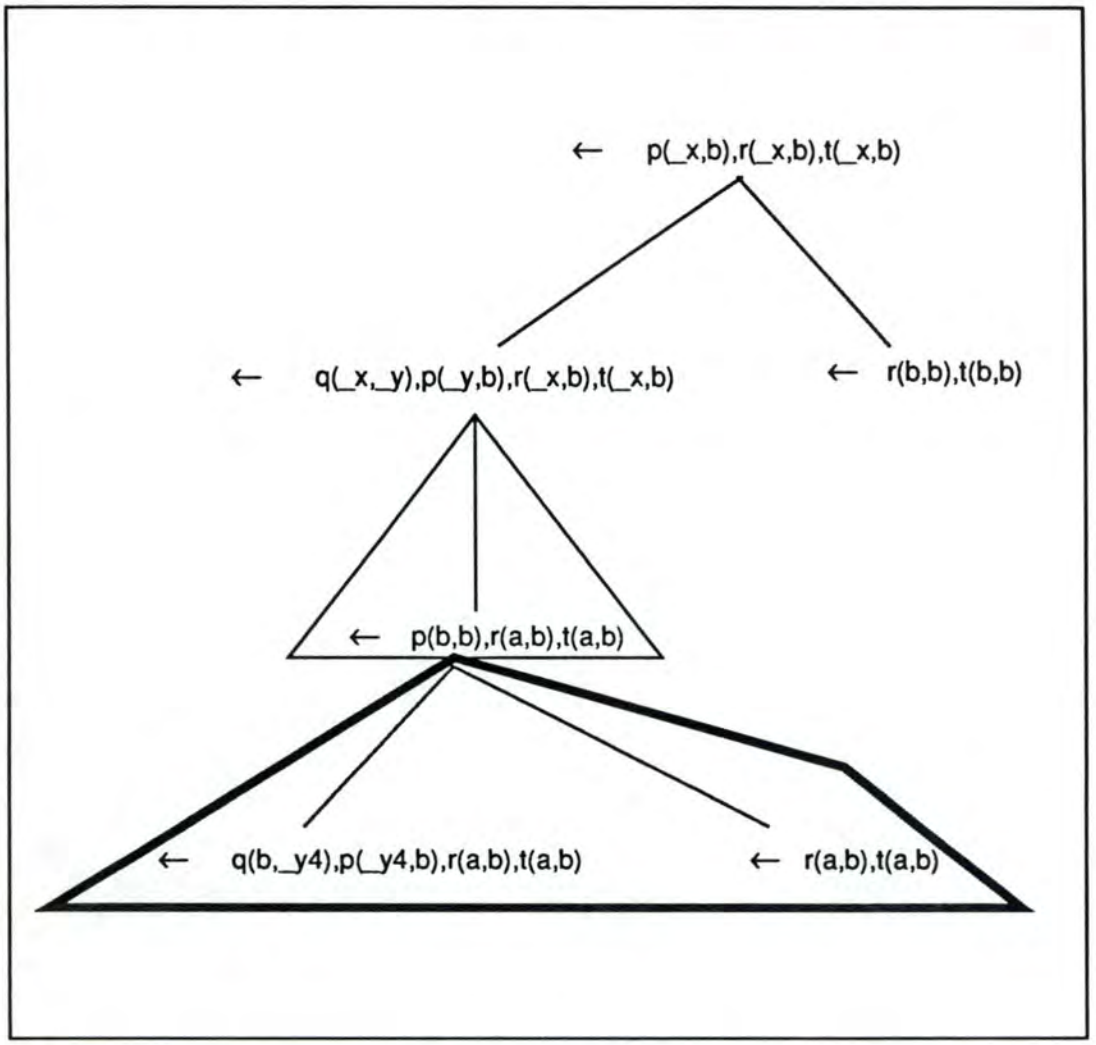

Figure 4.5

4.2.4 Proposition 4.2 :

Proposition :

let P be a program
let G be a goal

if T(G) is infinite , each infinite branch reaches at least one infinite SS1(T(G)) (this means it passes by the root of the SS). If it reaches many infinite SS1(T(G)), only one of them includes the rest of the infinite branch, starting at its root .

Proof :

For each infinite branch, we have that if there is no such SS1(T(G)), it is impossible for the branch to be infinite because the number of subgoals in a goal is finite and therefore, we cannot have an infinity of SS1(T(G)) levels!

It is also clear that there can only be one such SS1(T(G)) because the SS1(T(G)) are disjoint.

qed

4.2.5 Proposition 4.3 :

Proposition :

let P be a program
let G be a goal

if T(G) is infinite and if B is an infinite branch of T(G), the sequence of infinite SS reached by B and such that each of them contains the rest of B starting at its root is an infinite sequence. Moreover, the j^{th} infinite SS of the sequence is of degree j for T(G).

Proof :

By proposition 4.2, we know B reaches only one infinite SS1(T(G)) and which includes the rest of B, starting at its root. It is also the first infinite SS it crosses because SS of higher degrees are included in SS1 and

can not be reached yet !

Now, assuming B reaches a j^{th} infinite SS which includes the rest of B starting at its root and which is of degree j, does it reach a $(j+1)^{th}$ such SS of degree (j+1) ?

We denote S this j^{th} SS and RS its restriction !

S has the form given at figure 4.6, where C denotes the j^{th} degree complement of RS and B' the rest of B , starting at the root of S.

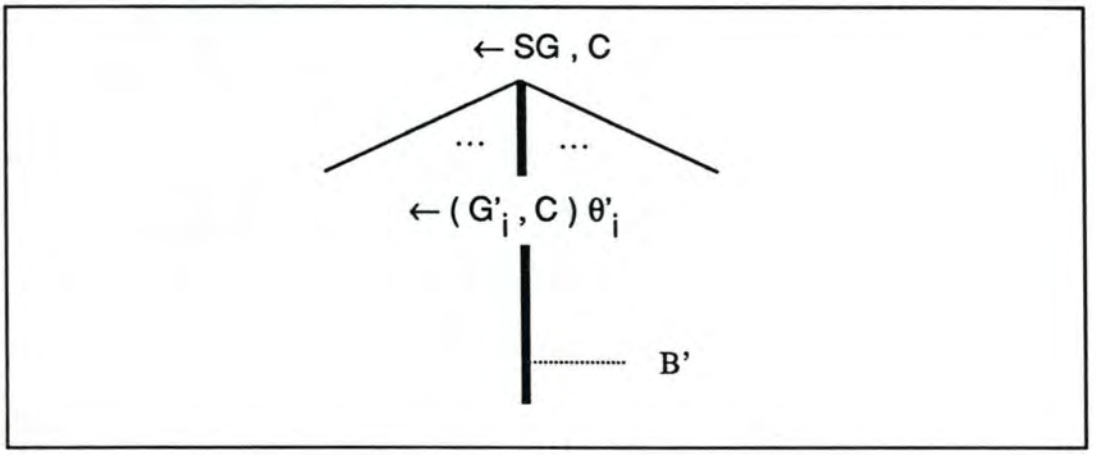

Figure 4.6

RS has the form given in figure 4.7, where RB' denotes the branch which becomes B', without its first arc, when we complete RS.

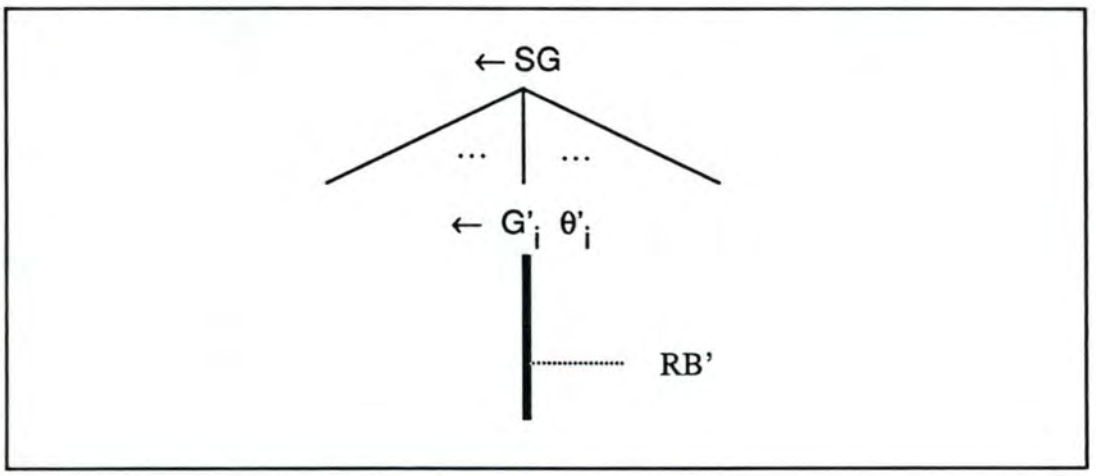

Figure 4.7

In $T(G'_i \ \theta'_i)$, the SLD-tree for $P \cup \{\leftarrow G'_i \ \theta'_i\}$, RB' is an infinite branch. From proposition 4.2, we know RB' reaches only one infinite $SS1(T(G'_i \ \theta'_i))$ which includes the rest of RB', starting at its root. It is also the first infinite SS, for $T(G'_i \ \theta'_i)$, it crosses because SS of higher degree for $T(G'_i \ \theta'_i)$ are included in $SS1(T(G'_i \ \theta'_i))$ and can not be reached yet !

We can associate a SS(j+1) for T(G) to the restriction of this SS1 for $T(G'_i \ \theta'_i)$ and it is clear that this SS(j+1) is the next one of the sequence of infinite SS for T(G) reached by B and such that each SS of the sequence includes the rest of B, starting at its root .

qed

4.3 STRUCTURE OF THE PROOF :

We begin to prove proposition 4.1 for SLD-trees containing one and only one infinite branch. The reason is that it is far more easier to understand ! For SLD-trees containing more than one infinite branch, we only suggest the reasonings.

In the case of SLD-trees with only one infinite branch, we first show three lemmas. Lemma 4.1 assert that for an incarnation of the algorithm EG created with parameters G and $V = V_0$ (such that $T(G \ \Theta V_0)$ contains one infinite branch) , there will be a succession of executions of this incarnation before an execution creates an incarnation of the algorithm ESG whose aim is to provide the CAS of the first infinite SS reached by the infinite branch (by proposition 4.3, this SS is of degree 1)and which contains the rest of the infinite branch, starting at its root. The j^{th} execution of this succession ends with $V = V_0 \ \theta_j$, θ_j being the j^{th} CAS for $P \cup \{\leftarrow G \ \Theta V_0\}$ such that its corresponding succes branch is more at left that any branch passing by the root of the first infinite SS reached by the infinite branch.

This first lemma corresponds in fact to the minimal case of lemma 4.3 which claims the same thing but for the k^{th} ($k \geq 1$) infinite SS reached by the infinite branch. But in order to prove lemma 4.3, the general case of the induction we make on k requires another lemma (lemma 4.2). In lemma 4.2, we show that if lemma 4.3 is correct up to (k-1), for an incarnation of algorithm ESG created with parameters SG

and $V = V_0$ (such that $T(SG\ \Theta V_0)$ contains one infinite branch), there will be a succession of executions of this incarnation before an execution creates an incarnation of the algorithm ESG whose aim is to provide the CAS of the k^{th} infinite SS reached by the infinite branch (by proposition 4.3, this SS is of degree k)and which contains the rest of the infinite branch, starting at its root. The j^{th} execution of this succession ends with $V = V_0\ \theta_j$, θ_j being the j^{th} CAS for $P \cup \{\leftarrow SG\ \Theta V_0\}$ such that its corresponding succes branch is more at left that any branch passing by the root of the k^{th} infinite SS reached by the infinite branch.

The proof for proposition 4.1 simply uses lemma 4.3 and proposition 4.3 .

4.4 PROOF FOR SLD-TREES CONTAINING ONE AND ONLY ONE INFINITE BRANCH :

4.4.1 Lemma 4.1 :

Lemma :

let P be the progam contained in CX
let G = SG1 , ... , SGn be a goal ($n \geq 1$)
let V_0 be an ACV covering G
let E0 be an incarnation of algorithm EG created with parameters G and $V = V_0$

assume that

- $T(G\ \Theta V_0)$ contains one and only one infinite branch, B .
- S*, a $kSS1(T(G\ \Theta V_0))$, is the first SS of the sequence of infinite SS reached by B and such that each of these SS includes the rest of B, starting at its root.
- S* is the 1SS1 for R which is thus a (k-1)SP1 for $T(G\ \Theta V_0)$.
- the root of R is $\leftarrow(SG_k , ... , SG_n)\ \Theta V_0\ \gamma_{k-1}$, γ_{k-1} being the

(k-1)SGS1 corresponding to the node on which R is fastened.

- S is the restriction of S* and is therefore a kSRS1 which is the SLD-tree for $P \cup \{\leftarrow SG_k\ \Theta V_0\ \gamma_{k-1}\}$.

if θ_1 , ... , θ_p ($p \geq 0$) is the sequence of answer substitutions defined by success branches located more at left than any branch passing by the root of S*, then

- the j^{th} execution of E0 ($1 \leq j \leq p$) ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- during the $(p+1)^{th}$ execution of E0, a first execution of an incarnation I1 of EG, created with (SG_k , ... , SG_n) and $V = V_0\ \gamma_{k-1}$ occurs. During this first execution, an incarnation I2 of the algorithm ESG is created with SG_k and $V = V_0\ \gamma_{k-1}$ and then, this execution of I1 passes at the point labelled *next_SG_CAS* .

Proof :

We use induction on k .

Case 1 : k = 1

This means S* is the 1SS1(T(G ΘV_0)) and R is the 0SS1(T(G ΘV_0)) which is T(G ΘV_0) itself. So, we can deduce that p = 0 ! It is clear that the incarnation I1 is in fact E0 and that during the first execution of EO an incarnation I2 is created with SG_k and $V = V_0$. This provides the thesis because $\gamma_0 = \varepsilon$

Case 2 : k > 1 , assuming it is true up to (k-1)

Sketch of T(G ΘV_0) :

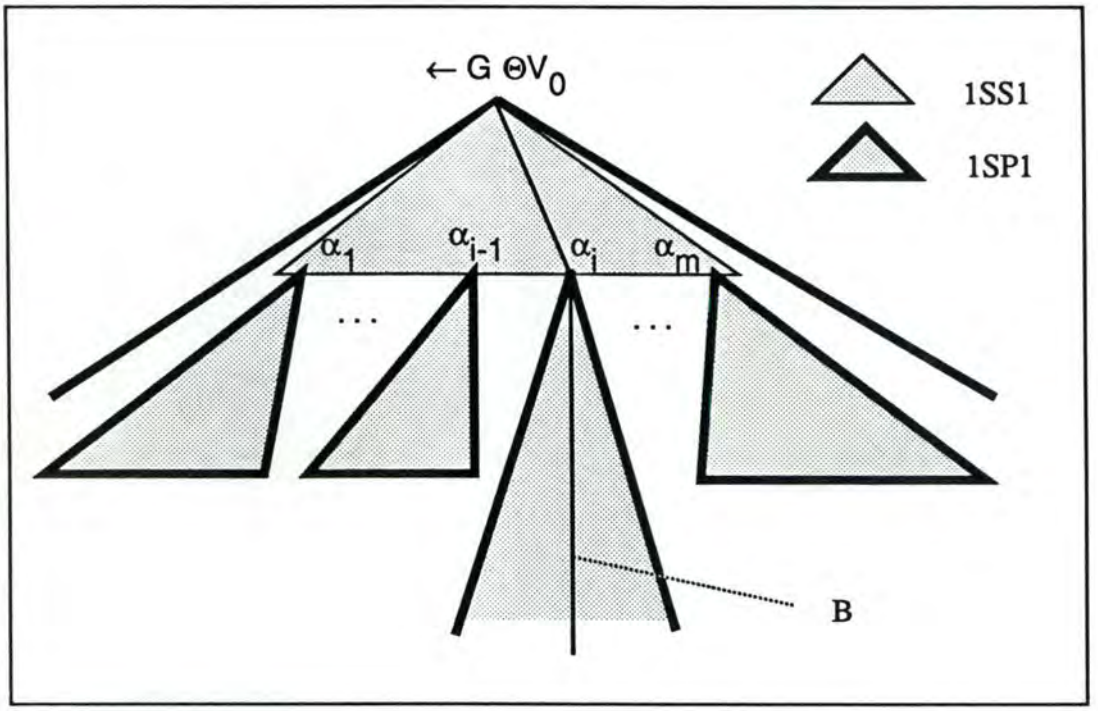

Figure 4.8

We know that the 1SS1 is finite (T(SG_1 ΘV_0) is finite). We denote α_1, ... , α_m the terminal nodes, of this 1SS1, which correspond to success nodes of the restriction 1SRS1 . We denote the corresponding CAS σ_1, ... , σ_m . These CAS are 1SGS1 !

Now, we can assume that θ_1 , ... , θ_{p1} ($0 \leq p1 \leq p$) is the sequence of answer substitutions such that for each of them, the respective success branch passes by one of the nodes α_1 , ... , α_{i-1} and that θ_{p1+1} , ... , θ_{p1+p2} ($0 \leq p2 \leq p\text{-}p1$) is the sequence of answer substitutions passing by the node αi but still more at left than any branch passing by the root node of S*.

We also have that the 1SP1 grafted to one of the first (i-1) grafting

nodes (α_1, ... , α_{i-1}) of the 1SS1 are finite SLD-trees. It is clear that the 1SP1 grafted to the i^{th} grafting node of the 1SS1 is the SLD-tree for P $\cup$ $\{\leftarrow (SG_2 , ... , SG_n)\ \Theta V_0\ \sigma_i\}$ and it contains an infinite branch B' which corresponds to the rest of B, starting at its root node. But in this SLD-tree , the first SS crossed by B' is a (k-1)SS1 with respect to $(SG_2 , ... , SG_n)$ $\Theta V_0\ \sigma_i$ (I).

Considering these remarks, we can use lemma 3.5 up to (i-1) to deduce that

- the j^{th} execution of E0 ($1 \leq j \leq p1$) ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- the $(p1+1)^{th}$ execution of E0 passes at the point labelled *next_SG_CAS* and the last execution of the incarnation E1 wil the i^{th} one.

But the incarnation E1 has been created with SG_1 and $V = V_0$. So, by proposition 3.1, the i^{th} execution that takes place at the point labelled *next_SG_CAS* ends with

$V = V_0\ \sigma_i$,
failure = false and
the incarnation E1 still alive.

So, the execution continues by creating an incarnation E2 of the algorithm EG with $(SG_2 , ... , SG_n)$ and $V = V_0\ \sigma_i$. But for this incarnation, lemma 4.1 is correct (due to (I)). Thus, it suffices to take over the same reasoning than for lemma 3.4 but using the induction hypothesis over lemma 4.1 rather than the assumption of correctness of proposition 3.2 for goals containing less than n subgoals in order to get the thesis !

qed

4.4.2 Lemma 4.2 :

Lemma :

let P be the program contained in CX
let SG be a subgoal
let V_0 be an ACV covering SG
let E0 be an incarnation of algorithm ESG created with parameters SG and $V = V_0$

assume that

- T(SG ΘV_0) contains one infinite branch B .
- S* is the k^{th} infinite SS, reached by B, which includes the rest of B, starting at its root node .
- S, the restriction of S*, is the SLD-tree for $P \cup \{\leftarrow SG' \gamma\}$.
- lemma 4.3 is true up to the $(k-1)^{th}$ infinite SS reached by an infinite branch and such that it includes the rest of this infinite branch, starting at its root.

if θ_1 , ... , θ_p ($p \geq 0$) is the sequence of answer substitutions for $P \cup \{\leftarrow SG \ \Theta V_0\}$ such that each of them is defined by a success branch located more at left than any branch passing by the root node of S* , then

- the j^{th} execution of E0 ($1 \leq j \leq p$) ends with
  $V = V_0 \ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- during the $(p+1)^{th}$ execution, an incarnation I1 of ESG is created with SG' and $V = V_1$ such that $\Theta V_1 = \gamma$.

Proof :

We suppose that T(SG ΘV_0) has the form :

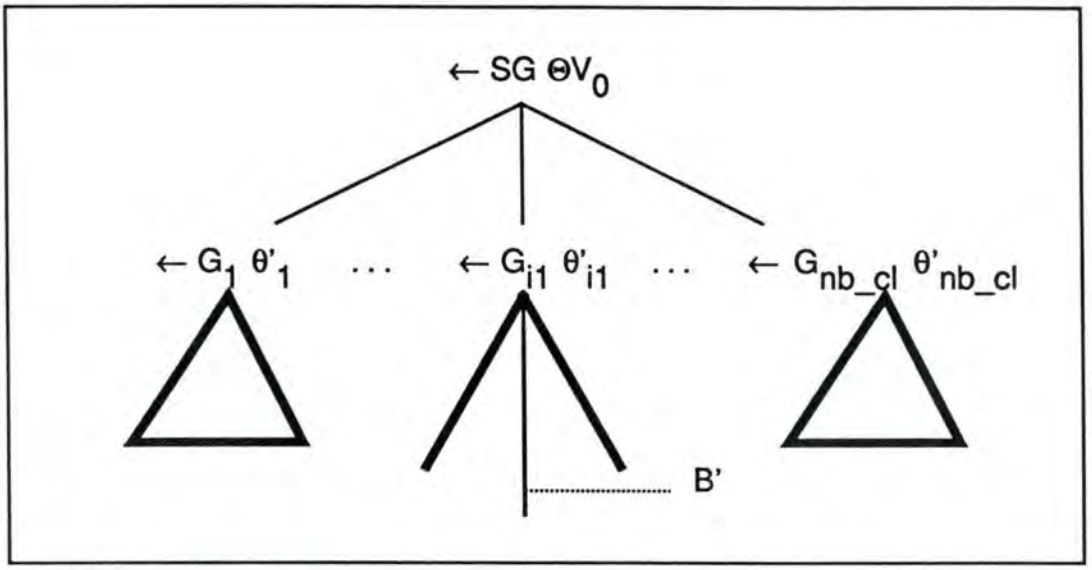

Figure 4.9

Where G_{i1} is the body of the $i1^{th}$ descendent input clause, θ'_{i1} a MGU of the head of this input clause and SG ΘV_0, and B' is the rest of B, starting at the node $\leftarrow G_{i1}\ \theta'_{i1}$.

From the definitions of SS, we can deduce that the SRS1(T($G_{i1}\ \theta'_{i1}$)) are SRS2(T(SG ΘV_0)). Following this reasoning, it is easy to see that SRSg(T($G_{i1}\ \theta'_{i1}$)) are SRS(g+1)(T(SG ΘV_0)) ! So, to the SS(k-1)(T($G_{i1}\ \theta'_{i1}$)), it corresponds a SSk(T(SG ΘV_0)) which can be obtained when we complete T($G_{i1}\ \theta'_{i1}$) with C θ'_{i1} where C is the first degree complement of the 1SRS1(T(SG ΘV_0)). But we can see that the 1SRS1(T(SG ΘV_0)) is in fact also the 1SS1(T(SG ΘV_0)) itself, so we have that C is the empty expression.

The first infinite SS reached by B and such that it includes the rest of B, starting at its root, is the 1SS1(T(SG ΘV_0)). As B' is the rest of B starting at node $\leftarrow G_{i1}\ \theta'_{i1}$, we have that the $(k-1)^{th}$ infinite SS for T($G_{i1}\ \theta'_{i1}$)

reached by B' and such that it includes the rest of B', starting at its root node, is a SS(k-1)(T(G_{i1} θ'_{i1})) (proposition 4.3). This SS(k-1)(T(G_{i1} θ'_{i1}) is therefore equivalent to the k^{th} infinite SS, reached by B, which includes the rest of B, starting at its root node. We have that the corresponding SRS(k-1)(T(G_{i1} θ'_{i1}) is the SLD-tree for $P \cup \{\leftarrow SG'\ \gamma\}$.

As the SLD-trees for $P \cup \{\leftarrow G_1\ \theta'_1\}$, ... , $P \cup \{\leftarrow G_{(i1-1)}\ \theta'_{(i1-1)}\}$ are finite, we can use lemma 3.3 and deduce that if θ_1 , ..., θ_{p1} ($0 \le p1 \le p$) is the sequence of answer substitutions for $P \cup \{\leftarrow SG\ \Theta V_0\}$ such that θ_j ($1 \le j \le p1$) is defined by a success branch passing by one of the nodes $\leftarrow G_1\ \theta'_1$, ... , $\leftarrow G_{(i1-1)}\ \theta'_{(i1-1)}$, then

- the j^{th} ($1 \le j \le p1$) execution of E0 ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- the $(p+1)^{th}$ execution of E0 passes at the point labelled *next_clause* with $i = i1$.

Now, this means that we have p2 ($0 \le p2 \le p\text{-}p1$) CAS $\theta_{(p1+1)}$, ... , $\theta_{(p1+p2)}$ for $P \cup \{\leftarrow SG\ \Theta V_0\}$ which form the sequence of CAS defined by success branches which passes by the node $\leftarrow G_{i1}\ \theta'_{i1}$ but are still located more at left than any branch passing by the root of S* !

So, given the hereabove remarks, by using, in place of proposition 3.2, lemma 4.3 for the $(k-1)^{th}$ SS, reached by B', which includes the rest of B' starting at its root, we can proceed to the same reasoning than for lemma 3.2 in order to deduce that

- the j^{th} execution of E0 ($p1+1 \le j \le p$) ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .

- during the $(p+1)^{th}$ execution, an incarnation I1 of ESG is created with SG' and $V = V_1$ such that $\Theta V_1 = \gamma$.

qed

4.4.3 Lemma 4.3 :

Lemma :

let P be the program contained in CX
let $G = SG_1 , ... , SG_n$ be a goal ($n \geq 1$)
let V_0 be an ACV covering G
let E0 be an incarnation of algorithm EG created with parameters G and $V = V_0$

assume that

- $T(G\ \Theta V_0)$ contains one infinite branch B .
- S^* is the k^{th} ($k \geq 1$)infinite SS, reached by B, which includes the rest of B, starting at its root node .
- S, the restriction of S^*, is the SLD-tree for $P \cup \{\leftarrow SG' \gamma\}$.

If $\theta_1 , ... , \theta_p$ is the sequence of answer substitutions for $P \cup \{\leftarrow G\ \Theta V_0\}$ such that each of them is defined by a success branch located more at left than any branch passing by the root node of S^* , then

- the j^{th} execution of E0 ($1 \leq j \leq p$) ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- during the $(p+1)^{th}$ execution, an incarnation I1 of ESG is created with SG' and $V = V_1$ such that $\Theta V_1 = \gamma$.

Proof :

We proceed by induction on k .

Case 1 : k = 1 .

S* is of degree 1 (proposition 4.3). In this situation, lemma 4.1 provides immediately the thesis .

Case 2 : k > 1 if it is true up to (k-1) .

If the first infinite SS, reached by B, which includes the rest of B, starting at its root, is the 1SS1(T(G ΘV_0)) :

Sketch of T(G ΘV_0) :

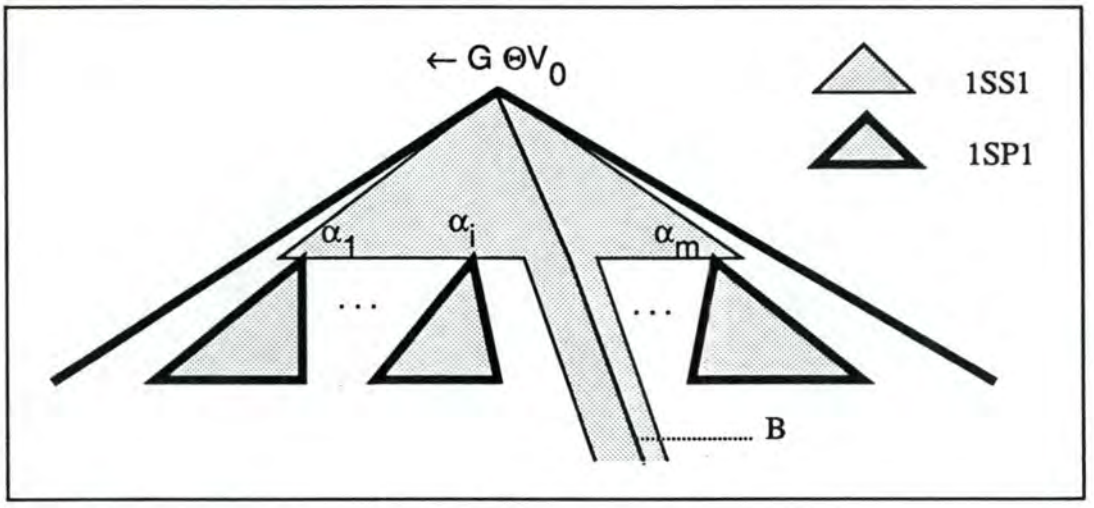

Figure 4.10

All the prolongations fastened on its grafting nodes are finite SLD-trees. So, with the same reasoning than for lemma 3.5 but using lemma 4.2 in place of proposition 3.1, we can deduce the thesis.

If the first infinite SS, reached by B, which includes the rest of B, starting at its root, is a gSS1(T(G ΘV_0)) (g > 1) , assuming the proof for lemma 4.3 has been given when the first infinite SS is a (g-1)SS1(T(G ΘV_0)) :

Sketch of T(G ΘV_0) :

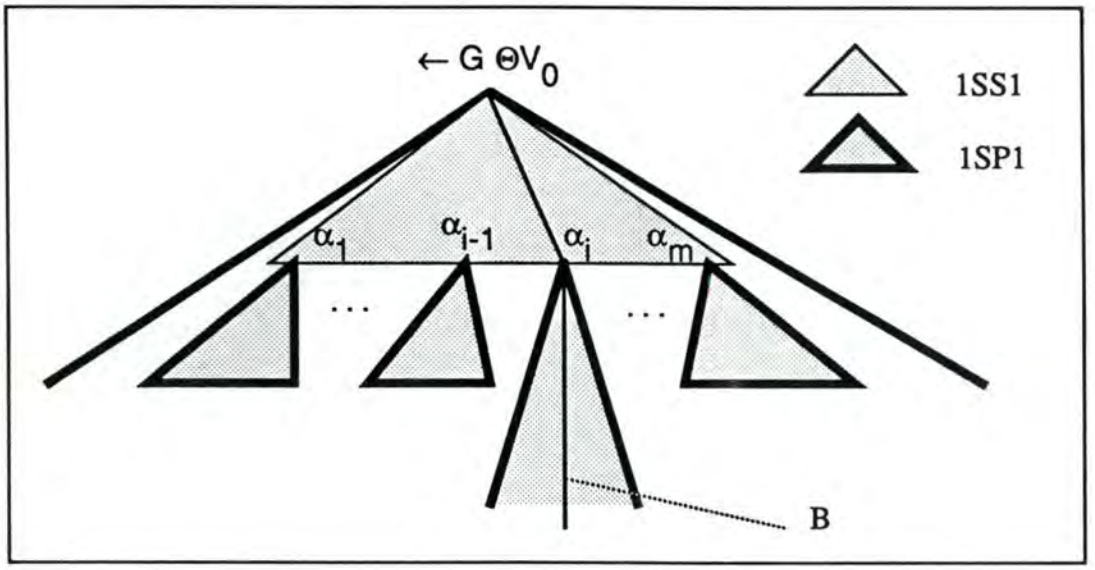

Figure 4.11

We know that the 1SS1 is finite (the SLD-tree for P ∪ {← SG_1 ΘV_0} is finite). We denote α_1 , ... , α_m the terminal nodes, of this 1SS1, which correspond to success nodes of the restriction 1SRS1 . We denote the corresponding CAS σ_1, ... , σ_m . These CAS are 1SGS1 !

Now, we can assume that θ_1 , ... , θ_{p1} ($0 \leq p1 \leq p$) is the sequence of answer substitutions such that for each of them, the respective success branch passes by one of the nodes α_1 , ... , $\alpha_{(i-1)}$ and that $\theta_{(p1+1)}$, ... , $\theta_{(p1+p2)}$ ($0 \leq p2 \leq p\text{-}p1$) is the sequence of answer substitutions passing by the node α_i but still more at left than any branch passing by the root node of S*.

We also have that the 1SP1 grafted to one of the first (i-1) grafting nodes of the 1SS1 are finite SLD-trees. It is clear that the 1SP1 grafted to the i^{th} grafting node of the 1SS1 is the SLD-tree for P ∪ {← (SG_2 , ... , SG_n) ΘV_0 σ_i} and it contains an infinite branch B' which corresponds to the rest of B, starting at its root node. But in this SLD-tree , the first SS crossed by B' is a (k-1)SS1.

Given these remarks, we can use lemma 3.5 up to (i-1) to deduce that

- the j^{th} execution of E0 ($1 \leq j \leq p1$) ends with
  $V = V_0\ \theta_j$,
  failure = false and
  the incarnation E0 still alive .
- the $(p1+1)^{th}$ execution of E0 passes at the point labelled *next_SG_CAS* and the last execution of the incarnation E1 was the $(i-1)^{th}$ one.

But the incarnation E1 has been created with SG_1 and $V = V_0$. So, by proposition 3.1, the i^{th} execution that takes place at the point labelled *next_SG_CAS* ends with

$V = V0\ \sigma i$,
failure = false and
the incarnation E1 still alive.

So, the execution continues by creating an incarnation E2 of the algorithm EG with $(SG_2 , \ldots , SG_n)$ and $V = V_0\ \sigma_i$. But for this incarnation, lemma 4.1 is correct . Thus, it suffices to take over the same reasoning than for lemma 3.4 but using the induction hypothesis over lemma 4.3 rather than the assumption of correctness of proposition 3.2 for goals containing less than n subgoals in order to get the thesis !

qed

4.4.4 Proof of proposition 4.1:

It comes straightforward from lemma 4.3 and proposition 4.3 . There is an infinity of infinite SS reached by the infinite branch and lemma 4.3 is true for any value of k .

qed

4.5 SLD-TREES WITH MANY INFINITE BRANCHES :

We do not provide a full proof for these SLD-trees. The reason is that it would be very long and a little bit tedious and we think it is convincing enough to underline the main points of the reasoning.

If the number of infinite branches is finite, it could proceed by induction over the number of infinite branches ! The reasoning should be quite the same than for SLD-trees with one infinite branch but centering on the leftmost infinite branch. However attention must be focused on a specific situation which is illustrated hereafter in the case of the proof for an SLD-tree with 2 infinite branches.

The problem is when the leftmost infinite branch B reaches a SS, S*, which is infinite but which does not include the rest of B, starting at its root !

Schematization of the restriction of S* :

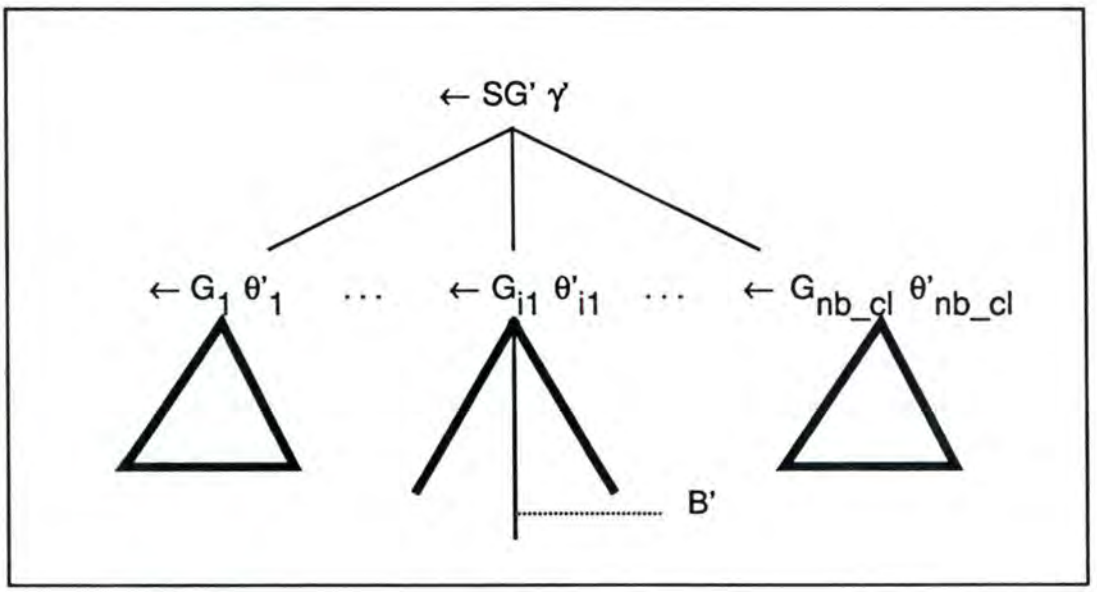

figure 4.12

Where G_{i1} is the body of the i1th descendent input clause, θ'_{i1} a MGU of SG' γ' and the head of this input clause , and B' is an infinite branch of $T(G_{i1}\ \theta'_{i1})$ which corresponds to the rest of the second infinite branch when the restriction is completed.

In this situation, we have that the leftmost infinite branch and the other

infinite branch follow the same way until the root node of S*. So, it means that the section, of the leftmost infinite branch, which is included in S* corresponds to a success branch of its restriction.

But, the restriction of S* contains only one infinite branch; so, when a such SS is encountered by the leftmost infinite branch, the use of proposition 3.1 must be replaced by the use of the induction hypothesis .

If there is an infinity of infinite branches, the reasoning should probably be based on the transfinites.

CHAPTER 5 : CUTS

5.1 INTRODUCTION :

The last thing we must prove is that our algorithms treat the cuts (!) in a right way !

Remember the cut is a widely used control facility of PROLOG. It is an extra-logical primitive having side-effects which can be described in terms of pruning in the SLD-tree. The next section recalls how this pruning occurs.

Then, we provide two propositions about algorithm ESG before we present how to fit the proof of equivalence for finite SLD-trees in order to take the cut into account. Finally, we turn to the problem of infinite SLD-trees.

5.2 SIDE-EFFECTS OF CUT :

If we ignore these side-effects, we can see the cut as a 0-ary predicate defined by the following fact :

!.

This means that it always succeeds one and only one time or in other words, the SLD-tree for $P \cup \{\leftarrow !\}$ contains one and only one CAS, CAS which in fact is the empty substitution ε . So, under this hypothesis, if GC is a goal obtained from a goal G simply by introducing cut(s) between some of the subgoals of G, we have that T(G) and T(GC) are different but their sequence of CAS are the same because ε is a left and right idendity for substitution composition.

But, due to the side-effects of cuts, the PROLOG-sequence can become different . We can imagine that the PROLOG sequence is still a depth first search sequence of CAS but of a *cutted SLD-tree* which is the SLD-tree where the pruning effects of cuts have been shown by deleting some of its parts.

To explain the pruning defined by a cut, we assume that there is a mechanism which permits us to uniquely name each cut. This can be done in much the same way than standardization of variables as it is explained in [Lloyd 84] : each cut is subscripted so that it becomes different from all other cuts of the clause where it appears but also from all other cuts already encountered in the derivation process.

Now, given a cut, we consider the node where it appears in the first place of the sequence of subgoals, the node where it appears for the first time and the parent node (if it exists) of this last one.

When the first node where the cut appears in the SLD-tree is the root (this node having no parent), the pruned part of the tree is composed of all the branches located more at right than the ones passing by the node where the cut appears in the first place of the sequence of subgoals.

When the first node where the cut appears is not the root, this node has a parent node that we call α . The pruned part consists of all the ends of branches, starting at α, and which are located more at right than the ends of branch, also starting at α but passing by the node where the cut appears in the first place of the sequence of subgoals.

To clarify this, consider the following program fragment taken from [Lloyd 84]:

```
A :- B , C .
  .
  .
  .
B :- D , ! , E .
  .
  .
  .
D .
```

where A, B, C , D and E are atoms.

Part of the SLD-tree for the goal clause "?- A ." is shown in the next figure :

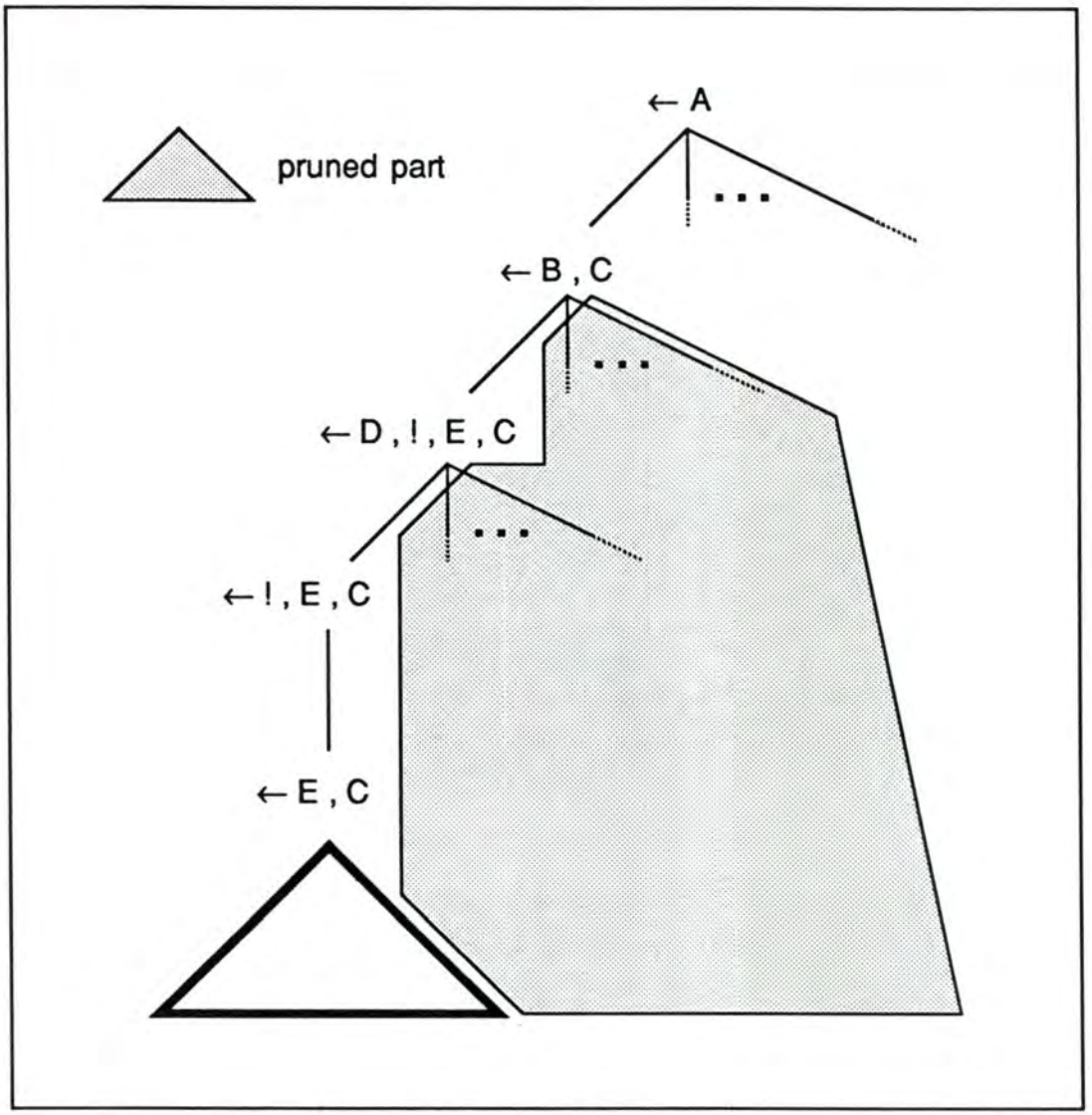

Figure 5.1

Note that if we consider the pruning for every cut appearing in the SLD-tree, it can happen that we get overlapping effects. This means that a cut implies the pruning of a part which is contained in a part pruned by another cut !

5.3 TWO PROPOSITIONS ABOUT ALGORITHM ESG :

5.3.1 Proposition 5.1 :

Proposition :

let P be a program contained in the context CX
let V_0 be an ACV
let E0 be an incarnation of algorithm ESG created with parameters SG = ! and V = V_0

- the first execution of E0 ends with
  V = V_0 ε = V_0 ,
  failure = false ,
  cut = true and
  the incarnation E0 still alive .
- the second execution of E0 ends with
  failure = true ,
  cut = true and
  the incarnation E0 destroyed .

Proof :

It comes immediately from a symbolic execution of the algorithm ESG .

qed

5.3.2 Proposition 5.2 :

Proposition :

any execution of an incarnation of the algorithm ESG that has been created with a subgoal different of the *cut* ends with cut = false.

Proof :

Let us begin by recalling the algorithm ESG when the subgoal is not the cut, showing the possible exit points of an execution :

```
if SG_instance ≠ "!" then
begin
        goto @ entry_pt ;

prem : i := 1 ;

next_clause : if  i > nb_cl  then      begin
                                              failure := true ;
                                              cut := false ;
                                              selfkill
                                       end ;
        CL' := STANDARDIZATION( CL_i , Vrem ) ;
                                { CL' = head(s'_i1 , ... , s'_in) :- G'_i }
        θ := MGU( CL' , SG_instance , Vrem ) ;
        Vnew := REFERENCE_ACV( CL' θ ) ;
        E1 := create( EG( G'_i θ , Vnew ) ) ;

next_CAS : execute( E1 ) ;
        if failure then  begin
                                if cut then begin
                                              cut := false ;
                                              selfkill
                                            end
                                else begin
                                        i := i + 1 ;
                                        goto next_clause
                                     end
                         end ;
        V := Vrem θ ΘVnew;
        cut := false ;
        terminate( next_CAS )
end
```

As all the exit points are preceeded by the instruction

cut := false

we immediately get the thesis !

qed

5.4 FINITE SLD-TREES :

We can keep most of the proof that has been developped in chapter 3. We consider in this section the needed adaptations for propositions, lemmas and their proofs ! We also add a new proposition (proposition 5.3) .

When we say that a cut (when it appears in the root node or, when it is not in the root, in one of the descendents of the root) stops the search of CAS, we mean that there is a node α , in the *cutted* SLD-tree, where this cut appears in first place of the list of subgoals. The pruning effects of this cut can be seen as a forced termination of the search for CAS within the branches located more at right than those passing by the same node α in the SLD-tree.

5.4.1 Adaptation of proposition 3.2 :

The second part of the thesis must be changed ! It becomes :

- if a cut appearing in G stops the search of CAS for $P \cup \{\leftarrow G \Theta V_0\}$

  then, the $(m+1)^{th}$ execution of E0 ends with
  failure = true ,
  cut = true and
  the incarnation E0 destroyed .

  else, the $(m+1)^{th}$ execution of E0 ends with
  failure = true ,
  cut = false and
  the incarnation E0 destroyed .

5.4.2 Adaptation of lemma 3.2 and of its proof:

Here also, the second part of the thesis must be splitted into two parts :

- if a cut appearing for the first time in the $i1^{th}$ descendent of $\leftarrow SG\ \Theta V_0$ stops the search of CAS

  then, the $(q+1)^{th}$ marginal execution of E0 ends with
  failure = true ,
  cut = false and
  the incarnation E0 destroyed .

  else, the $(q+1)^{th}$ marginal execution of E0 passes at the point labelled *next_clause* with $i = i1 + 1$.

In the proof, modifications must be brought to the second case, when we consider G'_{i1} is not the empty goal. Now, we use the hypothesis of correctness for the adapted proposition 3.2 . If a cut appearing for the first time in the $i1^{th}$ descendent of $\leftarrow SG\ \Theta V_0$, thus in $\leftarrow G'_{i1}\ \gamma_{i1}$, stops the search of CAS, we know from the adapted proposition 3.2 that the $(q+1)^{th}$ execution of E1 ends with
failure = true ,
cut = true and
the incarnation E0 destroyed .

So, in this situation, a symbolic execution of the instructions coming after this $(q+1)^{th}$ execution of E1 provides the thesis !

If no cut stops the search of CAS, the proof remains the same.

5.4.3 Adaptation of lemma 3.3 and of its proof:

The second part of the thesis is splitted into two parts :

- if a cut appearing in one of the G'k ($1 \leq k \leq nc$) stops the search for CAS

  then, the $(p+1)^{th}$ execution of E0 ends with
  failure = true ,
  cut = false and

the incarnation E0 destroyed .

else, the $(p+1)^{th}$ execution of E0 passes at the point labelled *next_clause* with $i = nc + 1$.

Case 1 of the proof remains the same provided we use the adaptation of lemma 3.2 !

For case 2, if no cut stops the search for CAS, the proof given in chapter 3 is still valid. If a cut appearing in G'_h ($1 \leq h \leq$ nc-1) stops the search for CAS, it means that the PROLOG subsequence of CAS passing by one of the nodes $\leftarrow G'_1 \ \gamma_1$, ... , $\leftarrow G'_h \ \gamma_h$, is in fact θ_1, ..., θ_p due to the side-effects of the cut (the branches passing by the descendents $\leftarrow G'_{(h+1)} \ \gamma_{(h+1)}$, ... , $\leftarrow G'_{nb\_cl} \ \gamma_{nb\_cl}$ are included in the pruned part). Considering the nc^{th} (nc > h) descendent of $\leftarrow$ SG ΘV_0 does not change anything because we know that the $(p+1)^{th}$ execution ends with

failure = true ,
cut = false and
the incarnation E0 destroyed .

So, we get the thesis .

Still for case 2, if the cut that stops the search of CAS appears in G'_{nc}, we can say there can not be any cut appearing in one of the nodes $\leftarrow G'_1 \ \gamma_1$,..., $\leftarrow G'_{(nc-1)} \ \gamma_{(nc-1)}$ which stops the cut. So, to get the thesis, it suffices to use the induction hypothesis and the adaptation of lemma 3.2 .

5.4.4 Proposition 5.3 :

Proposition :

an execution of an incarnation E0 of EG can not end with *cut* = true if no cut appears in the goal G received as parameter at creation.

Proof :

We proceed by induction on the number of subgoals composing G.

Case 1 : 0 subgoal .

A symbolic execution is enough.

Case 2 : n ($n \geq 1$)subgoals assuming it is true for (n-1) subgoals .

As we have seen in proposition 5.2, *cut* is never true after an execution of an incarnation of ESG for a subgoal different from !, even if a cut appearing in one of the input clauses for SG ΘV_0 stops the search for CAS within the SLD-tree for $P \cup \{\leftarrow SG\ \Theta V_0\}$. So, after execution of E1 (the incarnation of ESG created during the first execution of E0, with SG = first subgoal of G), if *failure* = true, a selfkill occurs and the execution of the incarnation E0 ends with *cut* = false because SG, the first subgoal of G, is not the cut. This covers one of the three possible exit points !

The two other exit points come in the tests following the execution of E2, an incarnation of EG created with G' which contains less than n subgoals . So, it never ends with *cut* = true because G' contains no cut. It is clear that only one of these two exit points can be reached in this situation and that *cut* = false.

qed

5.4.5 Adaptation of lemma 3.4 and of its proof :

First, we must assume that the adapted proposition 3.3 holds for the algorithm EG when treating goals of less than n subgoals.

Then, the third part of the thesis must be splitted into two subparts :

- if a cut appearing in G' stops the search for CAS
  then, the $(q+1)^{th}$ marginal execution of E0 ends with
  failure = true ,
  cut = true and
  the incarnation E0 destroyed .

  else, the $(q+1)^{th}$ marginal execution of E0 passes at the point labelled *next_SG_CAS* and the incarnation E2 is destroyed .

The frame of the proof given in chapter 3 is still correct provided we use the hypothesis of correction over the adapted proposition 3.2 . However, when j = q+1, we must consider a new situation when the $(q+1)^{th}$ execution of E2 ends with

failure = true ,
cut = true and
the incarnation E2 destroyed .

When this occurs, a symbolic execution of the instructions following this last execution of E2 easily provides the thesis, due to the value of *cut*.

5.4.6 Adaptation of lemma 3.5 and of its proof :

First, we must assume that the algorithm EG respects the adapted proposition 3.2 for goals containing less that n subgoals.

Then, the last part of the thesis must be splitted into two parts:

- if a cut appearing in G' stops the search for CAS
  then, the $(q+1)^{th}$ execution of E0 ends with
  failure = true ,
  cut = true and
  the incarnation E0 destroyed .

  else, the $(q+1)^{th}$ execution of E0 passes at the point labelled *next_SG_CAS* and the next execution of the incarnation E1 will be the $(i+1)^{th}$ one .

Note : recall that we consider the PROLOG-sequence of success nodes of $T(SG\ \Theta V_0)$. Recall also that proposition 3.1 talk about the PROLOG-sequence ! So, CAS for the first subgoal stump and which correspond to success nodes appearing in a pruned part of this stump are ignored by the incarnation, of ESG, E1 . If θ is a such CAS, it means that the prolongation $T(G'\ \Theta V_0\ \theta)$ is never considered by algorithm EG. So , possible answer substitutions for $P \cup \{\leftarrow G\ \Theta V_0\}$ corresponding to success branches ending in this prolongation are not considered. This is a correct behaviour in order to treat side-effects of cuts.

In the proof, the first case remains the same, provided we use the adapted lemma 3.4 .

For the second case, the proof given in chapter 3 holds when no cut stops the search of CAS. If a cut stops the search of CAS for $P \cup \{\leftarrow G'\ \Theta V_0\ \gamma_{i1}\}$ ($1 \leq i1 \leq p$), it means no such cut appears in the SLD-trees for $P \cup \{\leftarrow G'\ \Theta V_0\ \gamma_1\}$, ... , $P \cup \{\leftarrow G'\ \Theta V_0\ \gamma_{(i1-1)}\}$. So, if $i \leq i1$, the proof of the second case is still valid when using the adapted propositions and lemmas. But if $i > i1$, the PROLOG subsequence of answer substitutions for $P \cup \{\leftarrow G\ \Theta V_0\}$ such that, for each of them, the corresponding success branch passes by one of the grafting nodes of the first subgoal stump is an empty subsequence (this is due to the pruning of the tree by the cut). Recall that these grafting nodes are the nodes of the first subgoal stump corresponding to the j^{th} PROLOG success node of the restricted first subgoal stump . In this case, we can also say that for $i = j1$, we already have that

- the i^{th} execution of EO ends with
  $V = V_0\ \theta_i$,
  failure = false and
  the incarnation E0 still alive .
- the $(q+1)^{th}$ execution of E0 ends with
  failure = true ,
  cut = true and
  the incarnation E0 destroyed .

These conclusions remains true for $i > i1$ because E0 is destroyed.

5.4.7 Adaptation of the proof for adapted proposition 3.2 :

In fact the proof remains similar to the one developed in chapter 3 but it uses the adapted propositions and lemmas and a special case must be considered when the first subgoal is the cut and that no other cut has stopped the search . In this case, the (q+1)th execution of E0 must end with

failure = true ,
cut = true and
the incarnation E0 destroyed.

It is easy to deduce that by using proposition 5.1 after adapted lemma 3.5 .

5.4.8 Adaptation needed for lifting circularity :

We only need to add new minimal cases :

- depth(T(SG ΘV_0)) = 1 and SG = ! :

  this case is covered by proposition 5.1 .

- depth(T(G ΘV_0)) = 1 and G = SG , G' with SG = ! and G' = () :

  this case is covered by the adaptation of lemma 3.5 .

5.4.9 Proof of equivalence :

It does not need adaptations but we must use now the adapted lemmas and propositions !

5.5 INFINITE SLD-TREES :

Proposition 4.1 can not hold anymore !

Why ?

Simply because the SLD-tree can be infinite but the sequence of answer substitutions finite and such that the $(m+1)^{th}$ execution of E0 well ends because the infinite branch is in fact pruned by a cut !

So, the trick here is to use the same concepts and the same demonstrations but working on the *cutted* SLD-trees rather than on complete SLD-trees. This can work because we are in state to deduce from the previous section that our algorithms can compute the sequence of CAS for cutted SLD-trees when they are finite !

CHAPTER 6 : OUTLOOKS

6.1 INTRODUCTION :

Now that we have proved the equivalence between our procedural semantics and the usual one, we quickly give some ideas over possible future works using the concepts we have introduced.

In the next section, we examine the problem of the specification of logic procedures. The introduction of extra-logical features of PROLOG in our procedural semantics is briefly treated in section 6.3 before we turn to the question of the proof of correctness for PROLOG programs and the question of the occur check .

6.2 TOWARDS THE DEFINITION OF A FRAME OF SPECIFICATION :

The aim is not to provide a full discussion of this matter ; for a global study about specification, the reader should consult [Le Charlier 85]. We only see how some features of our procedural semantics can be used to specify accurately PROLOG procedures. The frame of specification that we introduce should facilitate reasoning when constructing and proving procedures. It is partly inspired from the work of Deville [Deville 87] which provides a good overview of the specification problem in logic programming. From our point of view, the changes we introduce to this frame allows to deal with a larger number of situations. However, we must admit that the logical aspects of a procedure fade.

Recall that a PROLOG procedure p of arity n is a sequence of program clauses having the same principal functor p, with arity n, in the head of each of these clauses.

Note : We do not pretend that the hereafter described frame of specification is the best there can be. We also think that it must not be perceived as a strait-jacket, this means it should be adapted if necessary !

6.2.1 General form of a specification :

We imagine the following form :

procedure_name(par_1 , ... , par_n)

Uses :

<1>
...
<i> Types
In-directionnality
Preconditions
[Relation]
Out-directionnality
Postconditions
...
<m>

The specification is aimed to provide informations about an incarnation of algorithm ESG created with SG = procedure_name(t_1 , ... , t_n) (where t_i , $1 \leq i \leq n$, is the i^{th} *argument*) and $V = V_0$. So, normally, we should speak in terms of executions of this incarnation but , by conventiion, we rather speak about *executions of the subgoal* SG ΘV_0 . This enables to be closer to the intuitive understanding. When we say that an execution of the subgoal succeeds (fails) , it means that the corresponding execution of the incarnation of ESG ends with *failure* = false (true). Each successful execution ends with $V = V_0 \theta$, where θ is a CAS for $P \cup \{\leftarrow SG \Theta V_0\}$ (if P is the program defined in the context CX).

This general form allows the description of each of the *m* possible uses of the procedure. A possible use describes the effects of the execution(s) of the incarnation (of the subgoal) if some conditions hold at the creation of the incarnation of ESG (equivalently, just before the first execution of the subgoal) . These conditions are described through the subdivisions *types* , *in-directionnality* and *preconditions*. Note that the possible uses must be mutually exclusive in order to avoid ambiguity. The effects of execution(s) are described through *types* , *relation* , *out-directionnality* and *postconditions* .

par_1 , ... , par_n are the *parameters* of the of the procedure. If we consider the subgoal SG ΘV_0 , we have that $par_i = t_i\ \Theta V$. Thus, its value evolves in time as V evolves. Just before the first execution of the subgoal, we have $par_i = t_i\ \Theta V_0$. After a successful execution ending with $V = V_0\ \theta$, $par_i = t_i\ \Theta V_0\ \theta$ due to proposition 2.2 and the fact that V_0 covers SG . In fact, par_1 , ... , par_n can be seen as boxes which retain the value of the actual parameters .

6.2.2 Types :

This passage is greatly inspired from [Deville 87] !

The content of the *Types* subdivision has the following general form :

let par_1 be a $type_1$
par_2 be a $type_2$
...
par_n be a $type_n$

where $type_i$ is the name of a *type* , a type being a set of ground elements (eg : integers, lists, trees, ...) .

We now define the set $type^*_i$ as the set of terms (ground or not ground) which have a ground instance belonging to $type_i$. More formally, let T be a term,

$$T \in type^*_i \Leftrightarrow \exists\, \theta : T\,\theta \in type_i$$

Examples :

$2 \in$ integer ,
[a,b] $\in$ list,
_x $\in$ integer*,
[a | _x] $\in$ list* .

(variables begin with an underscore)

The type of parameters has actually two different meanings. First, if par_i is not

a ground term (see section 6.2.3) just before the first execution , it gives information on the type of that parameter after a successful execution of the subgoal : $par_i \in type^*_i$. Second, types are preconditions to parameters. Just before the first execution of the subgoal, there must exist a substitution γ such that, $par_i \gamma \in type^*_i$ for at least one possible use . These preconditions are called *types preconditions.*

If these preconditions are not fulfilled for at least one possible use, we adopt the convention that the effect of execution is undefined. Another convention could be that the execution fails. But this would imply explicit type checking in the implementation.

6.2.3 In-directionnality :

The in-directionnality describes a form for each par_i . We retain three possible forms: *ground, free* and *partial* . These forms are the ones used in BIM_Prolog manual [BIM 86]

A parameter is said to be *ground* when it does not contain any variable. It is said to be *free* when it is only composed of variables for which bindings in V are of the form _v/_v and for which there is no binding of the form _y/_v in V. It is said to be *partial* if it is not ground and not free.

An in-directionnality is noted as

$$in(m_1, \ldots, m_n)$$

where

$$m_i \neq \{\} \quad (1 \leq i \leq n)$$

$$m_i \subseteq \{ground,free,partial\} \quad (1 \leq i \leq n)$$

For readability convenience, we denote each singleton {f} as f. We also define *any* as {ground,free,partial}.

We say that parameters $par_1, \ldots, par_n$ satisfy an in-directionnality

$$in(m_1, \ldots, m_n)$$

iff , just before the first execution of the subgoal, each par_i has one of the form of the set m_i . If the parameters do not satisfy to an in-directionnality of a possible use, this use is impossible for the inputs at hand. So, an in-directionnality consists of preconditions for a possible use. These preconditions are called *form preconditions*.

If these preconditions are not fulfilled for at least one possible use, we adopt the same convention than for types preconditions.

6.2.4 Preconditions :

This subdivision is used to describe other preconditions than type or form preconditions. These can be described in formal language but not necessarily.

Note that the preconditions for a possible use consists of the combination of those preconditions , the types preconditions and the form preconditions. This combination form what we call the *use preconditions*. If it is impossible to find a use for which the use preconditions hold, we adopt the convention that the effect of execution is undefined.

6.2.5 Relation :

This subdivision specifies a relation between the parameters. The aim of the procedure is to determine if this relation holds for the parameters. By relation, we mean a set of ground n-tuples $<a_1,a_2, \ldots , a_n>$. If the relation is not a simple one, some appropriate concepts as well as the relation itself must be defined accurately (possibly outside the specification).

This subdivision is optionnal . In fact, it should be used for procedure having a useful interpretation at the declarative level.

6.2.6 Out-directionnality :

The out-directionnality describes the form (see section 6.2.3 for the possible forms) of each par_i after a successful execution and the possible number of successful executions when the parameters comply to the use preconditions just before the first execution . The possible number of successful executions is specified by a lower and upper bound . The following values have been chosen for the lower and upper bound : a positive integer, infinite (∞) and a finite but

unknown positive integer (denoted by *).

An out-directionnality is noted as

$$out(M_1, \ldots, M_n) \text{ <Min-Max>}$$

where

$M_i \neq \{\}$ $(1 \leq i \leq n)$

$M_i \subseteq \{ground,free,partial\}$ $(1 \leq i \leq n)$

$Min \in N \cup \{\infty\}$ (N being the set of positive integers)

$Max \in N \cup \{*, \infty\}$

The meaning of the lower and upper bound to the number of successful executions requires perhaps some enlightment for the * value. It is useless as possible lower bound because it would be equivalent to a lower bound with value 0. As an upper bound, the value * means that the number of successful execution is finite. For instance, <2-*> means the number of successful execution is greater or equal to 2 but finite ! Note also that the actual number of executions is one greater than the upper bound if it is finite and the last execution fails . So, if the lower bound is 0, it means the first execution can fail.

It is obvious that these numbers provide information over the number of CAS for $P \cup \{\leftarrow SG \Theta V_0\}$ because it is equal to the number of successful execution .

6.2.7 Postconditions :

Here are specified other postconditions than those expressed via the *types*, *relation* and *out-directionnality* subdivisions. Specific characterization of the number of successful executions or of the results of each execution is are examples of what can be found here. For instance, if we know there can be more than 1 successful execution, we can specify that the j^{th} execution ends with $V = V_0\ \theta_j$ and provide some properties (usually depending on i) for θ_j or for the parameters .

The combination of these postconditions with the ones expressed via *types*, *relation* and *out-directionnality* forms what we call the *use postconditions* .

6.2.8 Examples of specification :

Note : When some use preconditions and/or use postconditions are valid for any use, they can be specified just before the beginning of the enumeration of possible uses !

erase(*x , list , list_erased)*

Types :

let *x* be a term
list and *list_erased* be lists

Relation :

the procedure determines whether *x* is an element of *list* and *list_erased* is *list* without the first occurrence of *x* in it .

Uses :

<1>

In-directionnality :

in(any , ground , any)

Out-directionnality :

out(ground , ground , ground) <0-*>

<2>

In-directionnality :

in(ground , free , ground)

Precondition :

$x \notin list\_erased$

Out-directionnality :

out(ground , ground , ground) <0-*>

Postcondition :

after the j^{th} execution, *x* appears in j^{th} position in *list*

append(*list1* , *list2* , *list_res*)

Types :

let *list1* , *list2* and *list_res* be lists

Relation :

the procedure determines whether *list_res* is the concatenation of *list1* and *list2*.

Uses :

<1>

In-directionnality :

in(ground, ground, any)

Out-directionnality :

out(ground , ground , ground) <0-1>

<2>

In-directionnality :

in(free , free , ground)

Out-directionnality :

out(ground , ground , ground) <m-m>

Postcondition :

m = (number of elements of *list_res* + 1)

after the j^{th} execution, *list1* is composed of the first (j-1) elements of *list_res* and *list2* of the rest

6.3 TOWARDS THE INTRODUCTION OF EXTRA-LOGICAL FEATURES :

The specification frame can also be used to specify the PROLOG built-ins, but some adaptations are needed because a lot of them cover extra-logical features (ie files, output devices) and work by side-effects. So, we must introduce a new type of preconditions : *environment preconditions*. These are described in a first new subdivision of the frame which is called *pre-environment* .

To express these preconditions, some concepts must be accurately defined. For instance, if we consider the built-ins working on ASCII files, we must define how to

characterized each file ! To do this, we can take over some ideas developed in [Derroitte 86].

A file is characterized by

- a logical name and a physical name.
- a status : open in reading, writing or extending mode
  closed (logical name undefined)
- a content : sequence of ASCII characters.
- an available content : suffix of the content (defined if open in reading mode).

We also define the current character of a file as the first character of its available content. We extend the characterization of the context so it includes the informations about the open files.

Now, a context is characterized by :

- a program.
- a set of open files of different logical names and of different physical names.

Now, the specification of side-effects that occur during some executions can be done in a second new subdivision : *post-environment* .

But we can use our procedural semantics in order to achieve an accurate specification of all the side-effects that can occur. If Min and Max are the bounds for the number of successful executions and if we create an incarnation E0 of ESG with SG = procedure_name(t_1 , ... , t_n) and V = V_0 (V_0 covering SG) , we can say there will be at least (Min+1) executions of E0 (if Min is finite) and at most (Max+1) executions of E0 (if Max is finite). It is possible with our frame to specify the side-effects for each of these executions if we want.

The possibility to specify the side-effects for each execution, coupled with the easiness to express tricky combinations of preconditions allows a powerful and accurate expression for the specification.

Note that for pure side-effects procedures, the expression of a relation can be omitted as such procedures have no logical meaning.

The two new subdivisions *pre-environment* and *post-environment* can be used to specify any procedure which can cause side-effects.

Examples :

fclose(*logical_name*)

Types :

let *logical_name* be an atom

Uses :

<1>

In-directionnality :

in(ground)

Out-directionnality :

out(ground) <1-1>

Post-environment :

the file referred by the logical name *logical_name* is closed. This means it is removed from the context !

get0(*logical_name* , *ascii_code*)

Types :

let *logical_name* be an atom
ascii_code be an integer

Uses :

<1>

In-directionnality :

in(ground , any)

Pre-environment :

there is a file of logical name *logical_name* with a non-empty available content

Out-directionnality :

out(ground , ground) <0-1>

Post-environment :

the current character of the file *logical_name* is removed from its available content. If *ascii_code* is not ground just before the first execution, it has , after the first execution, the value of the ascii code of the removed character. If it is ground, the execution

succeeds if *ascii_code* is equal to the ascii code of the removed character

<2>

In-directionnality :

in(any)

Pre-environment :

there is not any file of logical name *logical_name* with a non-empty available content .

Out-directionnality :

out(any) <0-0>

6.4 TOWARDS PROOFS OF CORRECTNESS:

To illustrate this matter, let us take the (classical) example of the append procedure ! We want it to comply to the specification given in section 6.2 . We propose the following text :

```
append([] , _list , _list) .
append([_x1 | _rest1] , _list2 , [_x1 | _rest3]) :-
                append(_rest1 , _list2 , _rest3) .
```

Hereafter, we provide a proof for the second possible use specified.. However, we think the proof given here is still too much linked to the text of our algorithms. So, future works should investigate this matter.

Let V be a VACV of value V_0 containing bindings _x/_x and _y/_y but not containing any other binding with _x or/and _y in its left part. So we can say that variables _x and _y are free with respect to V. Let t be a term such that t ΘV = t $\Theta V_0 = [e_1 , \dots , e_n]$ where e_i ($1 \leq i \leq n$) are terms.

The subgoal p(_x,_y,t) ΘV, with $V = V_0$, may be executed (n+2) times. We can prove that the j^{th} execution of this subgoal ($1 \leq j \leq n+1$) ends with

failure = false ,

$\_x = [e_1 , \dots , e_{i-1}]$,

$\_y = [e_i , \dots , e_n]$ and

the other bindings of V unchanged.

The $(n+2)^{th}$ execution ends with *failure* = true .

The proof proceeds by induction on n .

The creation of an incarnation E0 of ESG with SG = append(_x , _y , t) and V = V_0 is so that Vrem is initialized to V_0 .

Case 1 : n = 0 , so t ΘV_0 = = []

The only program clause whose head is unifiable with SG ΘV_0 is the first clause of the definition of *append*. So we can say that nb_cl = 1. Let θ be the MGU, we have θ = {_x / [] , _y / []}.

So, it is easy to see that the the first execution of the incarnation E0 passes at the point labelled *next_clause* with *i* = 1. But we have that G_1 (body of the first *append* clause whose head matches SG ΘV_0) is the empty goal so, we can deduce from lemma 3.2 that this first execution of E0 ends with *failure* = false and V = V_0 θ and that the second execution passes at the point labelled next_clause with *i* = 2 . As θ = {_x / [] , _y / []}, we have after the first execution _x = [] and _y = []. Now, as 2 > nb_cl, we have that the second execution ends with *failure* = true (by lemma 3.1) .

Case 2 : n > 0 if it is ok up to n-1

We have t ΘV_0 = [e_1 , … , e_n] .

Now, the two *append* clauses have their head unifiable with SG ΘV_0 . So, nb_cl = 2 .

We still have that the first execution of E0 passes at the point labelled *next_clause* with i = 1 . But we have that G_1 (body of the first *append* clause whose head matches SG ΘV_0) is the empty goal so, we can deduce from lemma 3.2 that this first execution of E0 ends with *failure* = false and V = V0 θ and that the second execution passes at the point labelled next_clause with *i* = 2 . We have after the first execution _x = [] and _y = [] because θ is the MGU of SG ΘV_0 and the head of the first clause and so, θ = {_x / [] , _y / []}.

Now, as 2 = nb_cl , the second execution continues and when it creates the incarnation E1 , it is easy to see that we have

$\theta = \{ \_x / [e_1 \mid \_rest1] , \_y / \_list2 , \_x1 / e1 , \_rest3 / [e_2 , \dots , e_n] \}$,

Vnew = [_rest1/_rest1 , _list2/_list2].

So, the incarnation E1 is created for G = append(_rest1 , _list2 , $[e_2 , \dots , e_n]$). But, the k^{th} ($1 \le k \le n$) execution of this goal ends with

failure = false ,

$V = V0\ \theta_k$

and in θ_k , we have that _rest1 = $[e_2 , \dots , e_{k-2}]$ and _list2 = $[e_k , \dots , e_n]$.

The $(n+1)^{th}$ execution of this goal ends with *failure* = true and *cut* = false . These are consequences of the induction hypothesis over the subgoal SG = append(_rest1 , _list2 , $[e_2 , \dots , e_n]$) .

Now, it is easy to see that the j^{th} ($2 \le j \le n+1$) execution of E0 ends with

failure = false ,

$\_x = [e_1 , \dots , e_{i-1}]$,

$\_y = [e_i , \dots , e_n]$ and

the other bindings of V unchanged.

The $(n+2)^{th}$ execution of E0 passes at the point labelled next_clause with $i = 3 > nb\_cl$; so this execution ends immediately with *failure* = true .

We think that some work is needed in order to develop a terminology which is usable for correctness proofs but which much less relies on the text of our algorithms. Moreover, in our example, we do not deal with a clause body composed of many subgoals. This simplifies of course many things !

However, if it happens that a clause body is of the form

$$B_1 , \dots , B_m$$

some more steps are needed in order to get the results of an execution of the goal $(B_1 , \dots , B_m)\theta$ where θ is the MGU of SG ΘV_0 and of the head of the corresponding clause. To do this, it should be possible to prove assertions that are true after the execution of the first subgoal and then, after the second subgoal and so on.

6.5 THE OCCUR CHECK :

For simplicity, we have supposed that the unification algorithm performs the occur check. However, many PROLOG systems provide it only as an explicit option, due to performance considerations.

In order to take account of this particularity, we only have to change the specification of the MGU function (see section 2.3.3). If necessary, we think it should also be possible to deal with the occur check problem when specifying procedures.

CONCLUSION

Our new procedural semantics for PROLOG is based on three simple algorithms. We think it is clearer than the usual procedural semantics which is explained in terms of search in a SLD-tree . From our point of view, it is also free of any ambiguity.

We proved that our semantics is equivalent to the usual one because they result in the same CAS in the same order..

A major advantage of our semantics is that it does not use the fuzzy concept of *backtracking*. This concept is often not well understood and this leads to carelessly built programs. Normally, our semantics should allow a programmer to fully understand how PROLOG procedures and programs are executed. This should enable him to improve the quality of his work.

However, we must admit that the logical aspects are not very stressed. But the focus we adopt on the operational aspects allows us to easily integrate the extra-logical features of PROLOG. From a practical point of view, it can be very interesting because professional PROLOG environments, like the BIM_Prolog [BIM 86] for instance, provide a lot of extra-logical features (database interface, windowing and graphics, ...).

Now, some work is needed in order to develop the specification issue but also to investigate the field of proofs of correctness of PROLOG programs. All this work should ideally go towards the elaboration of a methodology for PROLOG programming. But remember that a methodology does not solve the problem itself; the solving of the problem remains the programmer's role .

[Kowalski 74]
KOWALSKI R.A. : *Predicate Logic as a Programming Language*. IFIP 74, pp 569-574.

[Kowalski 79]
KOWALSKI R.A. : *Logic for Problem Solving*. Artificial Intelligence Series, North-Holland, Amsterdam 1979.

[Kowalski 82]
KOWALSKI R.A. : *Logic as a Computer Language*. In [Clark 82], pp 3-18.

[Le Charlier 85]
LE CHARLIER B. : *Réflexions sur le Problème de la Correction des Programmes*. Thèse de Doctorat, Université de Namur 1985 .

[Leroy 75]
LEROY H. : *La Fiabilité des Programmes*. Presses Universitaires de Namur, 1975.

[Leroy 78]
LEROY H. : *La Fiabilité des Programmes*. Ecole d'été de l'AFCET, Notes de cours, 1978.

[Lloyd 84]
LLOYD J.W. : *Foundations of Logic Programming*. Springer-Verlag, 1984.

[Mendelson 79]
MENDELSON E. : *Introduction to Mathematical Logic* . Van Nostrand, Princeton 1979.

[Robinson 65]
ROBINSON J.A. : *A Machine-Oriented Logic Based on the Resolution Principle*. J.ACM 12(1), January 1965, pp23-41.

[Winston 84]
WINSTON P.H. : *Artificial Intelligence*. Addison Wesley 1984.