A New Operational Semantics For Prolog

Kaninda Musumbu

A New Operational Semantics For Prolog

Kaninda Musumbu
LaBRI (UMR 5800 du CNRS),
Université Bordeaux 1, France
351, cours de la Libération, F-33.405 TALENCE Cedex

Abstract

The majority of abstract interpretation models defined for Prolog use abstract operations which do not have explicit counterpart in the SLD-resolution. We propose, in this paper, an operational semantics closely related to these models. We prove his equivalence with the SLD-resolution and then we use it to prove the consistency of one of abstract semantics presents in [7, 8, 9].

Key-words: Logique programming, SLD-resolution, static analysis, abstract interpretation, program semantics.

1 Introduction

The declarative character of the logic programming (and, in particular, of pure Prolog) grows and grows with many possibilities of optimizations. Those must be restful on automatizable static analysis methods and formally correct. The abstract interpretation [5] is the principal one of these methods. Its application to the logic programming has been proposed by many researchers (for example [2, 12, 6, 13, 4]). This work proposes semantics models known as " abstract' ' which define total propr ieties of programs (valid for an infinite of different executions). A semantics model must be proved consistent, i.e. it allows to deduct only correct properties (but possibly imprecise) of the programs. The proof of consistency of an abstract semantics model must be done by report to reference (or standard) semantic. In logic programming the semantics of reference is called SLD-resolution (see for example [11]). However, for reasons of practical applicability, many models of abstract interpretation for Prolog uses primitive operations which do not have explicit counterpart in the semantics of reference. The principal one of these operations is called *extension* in [2] and *meet* in [12]. This divergence makes difficult a direct proof of consistency of the abstract interpretation models. With an aim of realizing of such evidence of simple manner and convincing we propose, in this paper a *new* operational semantics not for standard pure Prolog [1], who allows to easily establish the link with the extension operation . We prove the equivalence of our operational semantics with the SLD-resolution. Lastly, we apply it for proving the consistency of abstract semantics suggested in [14] on which the algorithms of abstract interpretation are based, describe in [7, 8, 9]. It is applicable also to the models proposed in [2, 3, 12, 4].

This paper is organized as follows. Section 2 contains a recall of standard semantics of pure Prolog and we introduce some notations. In section 3, we present the abstract semantics one of which consistency should be proved. We define our new operational semantics that we call *instrumental semantic* in section 4 and the proof demonstrate its equivalence with that of section 2, is given. Lastly, we state, in section 5, the principal properties of instrumental semantics that we use then to prove the consistency of abstract semantics. Finally, a conclusion is given in section 6.

[1]Indeed, for a syntactic alternative of pure Prolog, called *set of the standardized programs.*

2 Standard Operational Semantics (SLD-resolution)

The standard operational semantics of Prolog is defined by the SLD-resolution. The acronym SLD stands for selecting a literal, using a Linear strategy, restricted to Definite clauses. SLD-resolution is a refinement of the resolution principle introduced by Robinson ([11]).

2.1 Notations

We suppose known the classical concepts of variable, term, atom, clause and logic procedure. The considered logic programs are suppose pure: without negative literals and extra-logic predicates. A logic programming problem may be defined as the problem of finding a substitution σ such that

$$P \models (A_1, A_2, \ldots, A_k)\sigma$$

is provable in some formal system, with $k > 0$, P a set of clauses, and $A_1, A_2, \ldots, A_k$ are atoms. P is called the program, $A_1, A_2, \ldots, A_k$ is the goal and σ is the answer substitution. Concepts of answer substitution is derived from general unifier are also supposes known. We note $mgu(t_1, t_2)$ the most general unifiers of t_1, t_2. we will note τ invertible substitutions (known also as the renaming).

2.2 Abstract Syntax

The abstract syntax of Prolog can be defined by the following grammar:

$$\begin{array}{lll}
P & \in & Programs \\
pr & \in & Procedures \\
c & \in & Clauses \\
b & \in & Goals \\
a & \in & Atoms\ (or\ Subgoals) \\
f & \in & Functors \\
p & \in & ProcedureNames \\
x_i & \in & ProgramVariables \\
y_i & \in & RenamingVariables
\end{array}$$

$$\begin{array}{lll}
P & ::= & <>\ \mid pr.P \\
pr & ::= & <>\ \mid pr.c \\
c & ::= & p(x_1\ ,\ \ldots\ ,\ x_n)\ \leftarrow\ b \\
b & ::= & <>\ \mid a.b \\
a & ::= & t_i = t_j \mid p(t_1\ ,\ \ldots\ ,\ t_n) \\
t & ::= & y\ \mid f(t_1\ ,\ \ldots\ ,\ t_n)
\end{array}$$

2.3 Derivation Tree associated to a sequence of atom:$Dt(SA)$

The standard operational semantics associates to any sequence of atoms SA a set of computed answer substitutions (noted $Cas(SA)$). This set is defined by the way of a derivation tree $Dt(SA)$. We suppose giving a reference program. $Dt(SA)$ is defined as follows:

- Each node is labeled by a sequence of atoms. Each edge is labeled by a substitution.
- the root of $Dt(SA)$ is labeled by SA. The level of the root is equal to 1.
- Let a node of level $i \geq 1$, labeled by SA'. The immediate descendants of this node are defined as follows:

1. If SA' is empty, it does not have there.
2. If SA' is of the form $A.SA''$, and $C_1, \ldots, C_m$ the clauses whose head is unifiable with A. Then, the node has m direct descendant (of level $i+1$). The sequence of atoms labeling the j-eme of these nodes is obtained as follows: let
   - H_j, SA_j the head and the body of C_j
   - τ_j, a renaming substitution of C_j variables by means of new variables,
   - σ_j a mgu of A and $H_j\tau_j$, not using variables other than those of A and $H_j\tau$.

   Then, $((SA_j\tau_j)SA'')\sigma_j$ is the sequence of atoms in question.

- the joining edges of considered nodes to its direct descendants are labeleds by substitutions σ_j.

2.4 Success and failure nodes, Correct substitution result

A *success node of* $Dt(SA)$ is a node labeled by an empty sequence of atoms. Let i, the level of this node and $\sigma_2, \sigma_3, \ldots, \sigma_i$ substitutions labeling the arcs edges leading to the ancestors of this node of levels $2, 3, \ldots, i$ respectively. We call *correct substitution result* associated to this success node, the substitution: $(\sigma_2 \ldots \sigma_i)_{/var(SA)}$.
A *failure node* is a node without descendants labeled by a no empty sequence of atom.
$Cas(SA)$ denotes, by definition, the set of correct substitutions results associated tothe success node of SA.

3 Abstract Semantics

3.1 Standardized programs, program's variables

In practice, the abstract interpretation of logic programs is simplified if we suppose the programs put in *normalized* form. This idea has initially been presented in [2]. Our concept of standardized programs differs that of [2] by the introduction of the concept of *program variable*. We will call *standard variables* and *standard substitutions*, the variables and substitutions used by the semantics of reference. For the normalized programs, we postulate the existence of an infinite sequence of variables: $x_1, \ldots, x_i, \ldots$ distinct from the standard variables and called program's variable. A normalized program is such as all head clause is of the form $p(x_1, \ldots, x_n)$ and any atom of one of the forms $x_{i_1} = x_{i_2} (i_1 \neq i_2)$, $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$ or $p(x_{i_1}, \ldots, x_{i_n})$ $(n \geq 0)$.

It is easy to translate a pure Prolog program into an equivalent normalized program..

Example 1
append([],X,X).
append([A|B],C,[A|D]):- append(B,C,D).

Normalized version

append(X1,X2,X3):- X1=[], X2=X3.
append(X1,X2,X3):- X1=[A|B], X4=A, X5=B, append(X4,X2,X5),X3=[X4|X5]).

Program substutition, noted θ, is a set of the form $\{x_{i_1} \leftarrow t_1, \ \ldots x_{i_n} \leftarrow t_n\}$ with the t_i are standard terms. This concept formalizes the intuitive idea that the variables $x_{i_1}, \ \ldots x_{i_n}$ of the program are bounded, at one time of execution, with the terms $t_1, \ \ldots t_n$. There defined as follows the application of a standard substitution σ is a program's substitution θ:

$$\theta\sigma \ = \ \{x_{i_1} \leftarrow t_1\sigma, \ \ldots x_{i_n} \leftarrow t_n\sigma\}$$

.

$Csub_D$ denotes the set of program's substitutions of domain D (above: $D = \{x_{i_1}, \ \ldots x_{i_n}\}$).

3.2 Abstract Substitutions, Abstract Domain

We call *abstract substitutions* objects representing the properties of program substitution. In the next, we consider a fixed family "a priori" of sets of abstract substitutions, indiced by the domain of program substitutions. We define, thus a *abstract domain* while associating any finite set of program variables D, with an inductive set, denoted by $Asub_D$. The symbol β denotes abstract substitutions. Formally, β represents a set of program substitutions denoted $Cc(\beta)$. The function $Cc : Asub_D \to \mathcal{P}(Csub_D)$ is called concretization function.

3.3 Abstract Operations

Let p, a procedure of a normalized program, β_{in}, an input abstract substitution for p, descriving a certain class of call of p.

The essential problem to solve is to determine for each clause:

$$p(x_1, \ldots, x_n) \leftarrow B_1, \ldots, B_m.$$

a list of abstract substitutions:

$$\beta_0, \beta_1, \ldots, \beta_m$$

representing the properties' variables of the clause at the different points preceding and following each atom $B_1, \ldots, b_m$, such as:

$$p(x_1, \ldots, x_n) \leftarrow \beta_0\ B_1\ \beta_1,\ \ldots,\ \beta_{m-1}\ B_m\ \beta_m.$$

We will temporaly make the assumption that we have an *Oracle* which for any input abstract substitution and any procedure provides an ouput abstract substitution descriving correctly the results.

The calculation of β_0, start from β_{in}, will be made grace to an *"extension"* function: $Ext_{D,D'} : Asub_D \to Asub_{D'}$ with ($D = \{x_1, \ldots, x_n\}$ and D' are the set of the clause's variables.)

Then, the computation of β_i can be decomposed as follows: initially, we compute an abstract substitution which reduces the domain of β_{i-1} (let D) to the variables appearing in B_i (or D'). For that we will need an *"restriction"* function:

$$Restr_{D,D'} : Asub_D \to Asub_{D'}$$

It is necessary, then, to replace by $x_1, \ldots, x_n$, the program variables $x_{i_1}, \ldots, x_{i_n}$, appearing in B_i, to obtain an input abstract substitution for the corresponding procedure to the sub-goal. This will be done by means of a *"renaming"* function

$$Chgvar_\gamma : Asub_D \to Asub_{D'}$$

(with $D = \{x_{i_1}, \ldots, x_{i_n}\}, D' = \{x_1, \ldots, x_n\}$ and $\gamma : D \to D'$ is a bijection)

the Oracle will then enable to know substitution β_{out} representing ouput concrete substitutions.

It is known that instantiations bring with the variables in $t_{i_1}, \ldots, t_{i_n}$ at the time of the execution of B_i will be simultaneousness propagates in the other terms. Thus, we must have an extension operation:

$$GExt_{D,D'} : Asub_D \times Asub_{D'} \to Asub_D$$

which will make it possible to defer these instantiations on the other variables If B_i is a *built-in*, the calculation of β_i could be done according to two functions, abstractedly giving by users, capable to execute the unifications. There are two possible cases: $x_{i_1} = x_{i_2}$ or $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$. According to case's, oracle will be replaces by:

$AI\text{-}Var : Asub_D \to Asub_D$ or $D = \{x_1, x_2\}$

$AI\text{-}Func : Asub_D \times F_{n-1} \to Asub_D$ or $D = \{x_1, \ldots, x_n\}$ [2]

[2] F_n is the set of the arite N functors.

3.4 Set of abstract tuplets

To complete this presentation, it remains to resolve the problem of Oracle, it be-A-statement to find a means of computing it. Let *Sat* denotes this oracle. which means a set of abstract tuplets of the form:

$$(\beta_{in}, p, \beta_{out})$$

such that $\beta_{in}, \beta_{out} \in Asub_D$ and $D = \{x_1, \ldots, x_n\}$ with $n > 0$ is the arity of p and p a predicate symbol appearing in the reference program, and for each couple (β_{in}, p) formed by an input abstract substitution and a procedure name, the oracle gives the corresponding output abstract substitution β_{out}. The abstract interpretation process allows for a *Sat* to calculate another set of abstract tuplets Sat': Let $C_1, \ldots, c_s$, the clause of procedure p. For each abstract substitution β_{in} we can abstractedly execute each clause $C_1, \ldots, C_s$, which will give $\beta^1_{out}, \ldots, \beta^s_{out}$ output abstract substitutions:

$$\beta_1, \ldots, \beta_s$$

We restrict these abstract substitutions to the variables of p, grace to the operation $Restr_{D,d'}$. Then, we takes the *"union"* of these substitutions.

$$Union : Asub_D \times \cdots \times Asub_D \to Asub_D$$

which gives another output abstract substitution β_{out}. It is intuitively obvious that if oracle (Sat) proposes a correct output abstract substitutions and if the various operations also calculate consistent results, the new set of tuplets Sat' calculated will be also consistent. Our abstract interpretation process defines a *transformation of sets of abstract tuplets* and we will prove that this transformation predictes correct properties of output substitutions.

3.5 Definition of Abstract Semantics

Formally, the problem of *"computing the oracle"* will be formulated as follows. We define, using the abstract operations, a transformation:

$$Teta : S - sat \to S - sat,$$

where *S-sat* designate the set of sets of abstract tuplets. The definition of $Teta$ uses 3 families of auxiliary functions respectively indexed by a symbol of predicate of the program of reference, a clause and a sequence of goals:

$$Tp(\bullet, p, \bullet) : Asub_D \times S\text{-}sat \to Asub_D,$$

$$TC(\bullet, C, \bullet) : Asub_D \times S\text{-}sat \to Asub_D,$$

$$TSB(\bullet, SB, \bullet) : Asub_D \times S\text{-}sat \to Asub_D,$$

We uses also 4 families of abstract operations derived of the operations identifies:

$ExtC(C, \beta) = CExt_{D,D'}(\beta)$
(where D is the set of the variables of the head and D' of all the variables of C),

$RestrC(C, \beta) = Restr_{D',D}(\beta)$,

$ExtB(B, \beta, \beta') = GExt_{D',D''}(\beta, Chgvar_{\gamma^{-1}}(\beta'))$ (D'' is the set of the variables in B),

$RestrB(B, \beta) = Chgvar_{\gamma}(Restr_{D',D''}(\beta))$.

the transformation $Teta$ is defined by the following equations:

$(p, \beta') : \beta' = Tp(\beta, p, Sat)\}$

$(p, eta) = union(\beta_1, \ldots, \beta_n)$,

$\beta_i = TC(\beta, C_i, Sat)$,

and $C_i, \ldots, C_n$ are the clauses of p.

$TC(\beta, C, Sat) = RestrC(C, \beta')$

$\beta' = TSB(ExtC(C, \beta), SB, Sat)$,

and SB is the body of C.

$TSB(\beta, (), Sat) = \beta$,

$TSB(\beta, B.SB, Sat) = TSB(\beta_3, SB, Sat)$

$$\begin{array}{lll}
\text{with} & \beta_1 & = RestrB(B, \beta), \\
 & \beta_2 & = Sat(\beta_1, p) \qquad \text{if B is of form} \quad p(\cdots), \\
 & & = AI\_Var(\beta_1) \qquad '' \qquad\qquad x_{i_1} = x_{i_2}, \\
 & & = AI\_Func(\beta_1) \qquad\qquad\qquad x_{i_1} = f(\cdots), \\
\textit{and} & \beta_3 & = ExtB(B, \beta, \beta_2).
\end{array}$$

The abstract semantics of the program is, by definition, the smallest fix-point of $Teta$. Its existence is assured if we impose that the $Asub_D$ sets are inductive and that the primitive operations are monotonous. However, these properties is not necessary to prove the *consistency* of the semantics. We will thus not be delayed, here.

4 Instrumental Operational Semantics

4.1 Motivation

According to the operational semantics of reference, any execution of a program returns *" to raise a query Q"* of form: $p(t_1, \ldots, t_n)$ where p is the name of a program procedure and $t_1, \ldots, t_n$ are terms being able to contain standard variables. The results of the program constitute a set of substitutions having for domain the set of the variables appearing in $t_1, \ldots, t_n$. Intuitively, the objective of the algorithms of abstract interpretation is to compute *" properties"* of the results from certain assumptions on the form of the queries. These properties and assumptions are formalized by the concept of abstract substitutions. What means the *consistency* of abstract semantics? According to this semantics, a Prolog program defines a set of abstract tuplets: Sat. On the one hand, the condition:

$$(\beta, p, \beta') \in Sat$$

means: if θ is a substitution which verifies the *" property"* β and if σ is a result for the query $p\theta$, then, $\theta\sigma$ verifies the *" property"* β'.

The fact of verifying a property is formalized by the concretization function Cc. In addition, we are agreed to note $Cas(Q)$ the results' set for a query Q. Therefore, the correction of our semantics will be precisely as follows:

$$\theta \in Cc(\beta) \; et \; \sigma \in Cas(p(x_1, \ldots, x_n)\theta) \Rightarrow \theta\sigma \in Cc(\beta').$$

To be rigorous we will base our proof on the operational semantics of reference recalled in section 2, in a formulation slightly different of that of [11], destinated to provide an useful terminology and notations. A difficulty of the proof comes owing to the fact that the standard semantics is enough distant of the procedural model on which our abstract is a semantics copy. The idea of our proof is to show that for any number $k > 0$, if we limit the number of calls of overlaping procedures to k at most, the provided answers verifying the properties described by the abstract semantics. However, in the reference semantics, the structure of the calls of procedures in progress does not appear. To solve this problem, we propose now, a new operational semantics, we call *instrumental* that we prove its equivalence with that of section 2. Section 5 will be then devoted to prove the consistency of abstract semantics compared to the instrumental semantics.

Let us finish this section by presenting intuitively this semantics. The standard semantics defines a research tree for any query Q. Each node of this tree is labeled by a sequence of atoms SA and the computaton of the results consists to traverse this tree in-depth first. In another manner, we can see, the computation of the results like a triggering of call procedures of other call procedures. Let us suppose that the initial goal is of the form $p(x_1, \dots, x_n)\theta$, and fix one moment of the execution. According to the first semantics, one can be on a node labeled by SA. According to the second, there is already triggering , without having finish them, k procedures call: $p_1, p_2, \dots, p_k$. For each of those calls, a clause is in the course of execution and it remains to execute a sequence of sub-goals after execution of the current goal. Moreover, variables of the clause are "instantiated" to certain values which can be represented by program substitutions : $\theta_1, \dots, \theta_k$. The link between the two points of view is that SA can be obtained by concatenating the sequences of sub-goals:

$$SB_k\theta_k, SB_{k-1}\theta_{k-1}, \dots, SB_1\theta_1.$$

This sequence of sequences of defined sub-goals represents a structuration of the sub-goal running which is not taken into account by the classical definition of Lloyd. Our "new" definition corresponds to incorporate this structure to that of Lloyd. The goal SA will be there replaces by an object of the form:

$$((SB_k, \theta_k), (SB_{k-1}, \theta), \dots, (SB_1\theta_1)).$$

which we will note SE and call *resolvent.*

4.2 Preliminaries

The instrumental semantics is defined for the standardized programs. We call *element of resolvent*, any couple of the form (SB, θ) where SB is a sequence of standardized atoms and θ is a program substitution. A *resolvent* is a nonempty sequence of resolvent's elements. We note E and SE the elements of resolvent and the resolvent. A empty *resolvent* is one of the form $((), \theta)$.

4.3 Derivation tree associated to a resolvent:$Di(SE)$

Let SE a resolvent. Supposes fixed a normalized program of reference.
$Di(SE)$ is defined as follows:

- Each node is labeled by one resolvent and has a level. The edges are not labeled.
- the root of $Di(SE)$ is labeled by SE. Its level is 1.
- Each node of level $i \geq 1$, labeled by SE', its immediate descendants are defined as follows:
  1. If SE' is empty, it does not have.
  2. If $SE' = ((), \theta).SE"$ and $SE"$ is a resolvent (not empty sequence),
     there is only one successor, labeled by$SE"$, its level is i (and not $i+1$).
  3. If $SE' = (B.SB, \theta).SE"$ and $SE"$ is a resolvent or an empty sequence,
     there are two sub-cases to consider.
     (a) B is a built-in of form $t_1 = t_2$.
         If $t_1\theta$ and $t_2\theta$ are not unifiable,then there are no descendants.
         Else, let σ=mgu(t_1, t_2) using only variables of t_1, t_2. There is only one successor, of level $i+1$, labeled by $((SB, \theta).SE")\sigma$ with the notation $SE\sigma$ is defined by:
         $()\sigma = (); ((SB, \theta).SE)\sigma = (SB, \theta\sigma).SE\sigma$.

(b) B is an atom of the form $p(x_{i_1}, \dots, x_{i_n})$.
and $C_1, \dots, c_m$, the clauses of p. There is m successors. The j-eme of them is labeled by:
$(SB_j, \theta_j).(SB, \theta).SB$",
where SB_j is the body of C_j and
$\theta_j = \{x_1 \leftarrow xi_1\theta, \dots, x_n \leftarrow x_{i_n}\theta, x_{n+1} \leftarrow y_1, \dots, x_{n+p} \leftarrow y_p\}$
$var(C_j) = x_1, \dots, x_{n+p}$,
$y_1, \dots, y_p$ is fresh variables of renaming
All these successors are of level $i + 1$.

4.4 Definitions

A *success node of* $Di(SE)$ is a node labeled by an empty resolvent, let $((), \theta)$. θ is the *substitution result* associated to this success node. A *failure node* is a node without descendants labeled by an empty resolvent. One notes $Cas(SE)$, the set of substitutions result of $Di(SE)$.

4.5 Limited derivation tree associated to a resolvent: $Di_k(SE)$ $(k \geq 1)$

We define $Di_k(SE)$ as $Di(SE)$ with the following modification: If SE contains k elements, it does not have successors.

4.6 Properties

We prove aisely the following properties:

1. $Di_k(SE)$ is finished, for any k.
2. Let T a tree. We calls *"stump"* of T, any tree T' such as:
   - the root of T' is that of T;
   - if $s \in T'$, we have also $s \in T$ and the path from the root to s is the same one in T and T'.

   We have:

   - $Di_k(SE)$ is a stump of $Di(SE)$,
   - $Di_k(SE)$ is a stump of $Di_{k'}(SE)$, if $k \leq k'$.
3. $Cas_k(SE) \subseteq Cas_{k'}(SE) \subseteq Cas(SE)$ (si$1 \leq k \leq k'$).
4. $Cas(SE) = \bigcup_{k=1}^{\infty} Cas_k(SE)$.

4.7 Equivalence of both operational semantics

4.7.1 Definition: *Value of a resolvent* $(v(SE))$

We define by induction on the structure of the resolvent, the function $v(SE)$ which gives the next atoms represented by a resolvent:

$v((SB, \theta)) = SB\theta$,

$v((SB, \theta).SE) = SB\theta.v(SE)$,

Levels	**Nodes of** $Ar(p(u))$	**Nodes of** $Ai((p(x_1), \{x_1 \leftarrow u\}))$
1	$p(u)$	$((p(x_1)), \{x_1 \leftarrow u\})$
2	$q(v,u), v = a$	$((q(x_2, x_1), x_2 = a), \{x_1 \leftarrow u, x_2 \leftarrow v\}), ((), \{x_1 \leftarrow u\})$
3	$v = u, v = a$	$((x_1 = x_2), \{x_1 \leftarrow v, x_2 \leftarrow u\}), ((x_2 = a), \{x_1 \leftarrow u, x_2 \leftarrow v\}), ((), \{x_1 \leftarrow u\})$
4	$u = a$	$((), \{x_1 \leftarrow u, x_2 \leftarrow u\}), ((x_2 = a), \{x_1 \leftarrow u, x_2 \leftarrow u\}), ((), \{x_1 \leftarrow u\})$
		$((x_2 = a), \{x_1 \leftarrow u, x_2 \leftarrow u\}), ((), \{x_1 \leftarrow u\})$
5	empty nodes	$((), \{x_1 \leftarrow a, x_2 \leftarrow a\}), ((), \{x_1 \leftarrow a\})$
		$((), \{x_1 \leftarrow a\})$

Figure 1: Correspondence of operational semantics

4.7.2 Lemma

Suppose given a normalized program. Let SA a sequence of atoms, SB a sequence of normalized atoms, θ a program substitution such that $SA = SB\theta$ and $dom(\theta) = var(SB)$. There exists a subjective function from the set of the nodes of $Di(SB, \theta)$ in those of $Dt(SA)$, verifying the following property:
For all couple(N,N') of nodes put in correspondence:

- N and N' does have the same level,
- if N is labeled by SE then N' is by $v(SE)$,
- if i is the level of N and N' and

  $SE = SE'(SB'\theta')$,

  $\sigma_2, \sigma_3 \ldots, \sigma_i$ are substitutions labeling the edges carrying out from the root to N',

  then, $\theta' = \theta\sigma_2\sigma_3 \ldots \sigma_i$.

Proof Let i, any naturel number, one shows by recurrence on i, that such a function exists for the nodes of level i. One will find a complete proof in [14].

4.7.3 Theorem

Under the same conditions: we have:

$$Cas(SA) = \{\sigma : dom(\sigma) \subseteq var(SA) \ et \ \theta\sigma \in Cas(SB, \theta)\},$$

$$Cas(SB, \theta) = \{\theta\sigma : \sigma \in Cas(SA)\}.$$

Proof Consequence of the lemma (see [14]).

Example Let the normalized program

$p(x_1) \leftarrow q(x_2, x_1), x_2 = a.$
$q(x_1, x_2) \leftarrow x_1 = x_2.$

We choose for SA the atom $p(u)$ et for (SB, θ) the resolvant element $(p(x_1), \{x_1 \leftarrow u\})$ The derivation trees $Dt(SA)$ and $Di((SB, \theta))$ correspond according to methods' of lemma 4.7.2. They have only one branch whose various nodes are represented with the figure 1. (the letters u,v represent standard variables.)

5 Consistency of the abstract semantics

The principle of the proof is to show, by induction on k, that all substitutions results appearing in a derivation tree $Di_k(p(x_1, \ldots, x_n), \theta)$ verifying the property $Sat(\beta, p)$ if θ verify β and if Sat is a fix-point. For that it is useful to initially establish a certain number of properties of operational semantics. We state these properties with there proof in [14].

6 Conclusion

We have proposes in this paper a new operational semantic, called instrumental allowing to justify simply and rigorously the semantic models of abstract interpretation of Prolog. We prove the equivalence with operational semantics of reference [11] and then we use it to prove the consistency of a semantics model of abstract interpretation (used by [7, 8, 9]). It is also applicable to the models proposed in [2, 4, 12]. The abundance of current research in static analysis of the logic programs will lead to define new models more elaborates. These new models will be able, think, being justify while following the same proccess.

References

[1] **Mr. Bruynooghe** *A framework for the abstract interpretation of logic programs*, Carryforward CW62, Katholieke Universiteit, Leuven, 1987.

[2] **Mr. Bruynooghe et al..** *Abstract interpretation: Total Towards the optimization of logic programs*, In Proc. 1987 Symp. one Logic Programming, IEEE Society Press, pp 192-204.

[3] **Mr. Bruynooghe** *A practical framework for the abstract interpretation of logic programs*, Newspaper of Logic Programming, 1991.

[4] **P. Codognet, G File'** *Computations, abstractions and constraints for Logic Programs*, In Proc ICLP 91, June 91.

[5] **P. Cousot, R. Cousot** *Interpretation Abstract: In Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints*, POPL 1977, Sigact Sigplan, pp 238 – 252.

[6] **T Kanamori, T Kawamura** *Analysing success patterns of Logic Programs by Abstract Hybrid Interpretation*, Technical carryforward, ICOT, 1987.

[7] **B. Charlier, K Musumbu, P. Van Efficient Hentenryck** *and accurate algorithms for the Abstract Interpretation of Prolog Programs*, Internal report, Institute of Data processing, F.U.N.D.P. Namur, August 1990.

[8] **B. Charlier, K Musumbu, P. Van Hentenryck** *A generic abstract interpretation algorithm and its complexity analysis*, In Proc ICLP 91, June 91.

[9] **B. Charlier, P. Van Hentenryck** *Experimental of Evaluation has Generic Abstract Interpretation Algorithm for Prolog.* In Proc ICCL 92, San Francisco, April 1992.

[9] **Roberto Barbuti, Michael Codish, Roberto Giacobazzi and Michael Maher** *Oracle Semantics for Prolog*

[10] nformation and Computation; 22 (2): 178-200, 1995

[11] **J.W. Lloyd** *Foundation of Logic Programming*, Springer Verlag, 1987.

[12] **K Marriott and H. Sondergaard** *Notes for has Tutorial one Abstract Interpretation of Logic Programs*, NACLP 89, Cleveland, 1989.

[13] **C.S. Mellish** *Abstract Interpretation of Prolog Programs*, In Abstract Interpretation of Declarative Languages, Ellis Horwood Limited, 1987.

[14] **K Musumbu** *Interprétation abstraite des programmes Prolog* , Thèse de doctorat, Institut d'Informatique, F.U.N.D.P., Namur, septembre 90.

[15] **J.E. Stoy** *Denotational semantics: The Scott-Strachey Approach Tom programming Language Theory*, MIT Press, Cambridge, Mass., 1977.