Accurate Analysis of Prolog with cut

Giorgio Levi and Fausto Spoto
Dipartimento di Informatica
Università di Pisa
{levi,spoto}@di.unipi.it

Abstract

We present a new PROLOG semantics which accurately models the operational behaviour of some of the control features of PROLOG, namely the search rule and the cut. The semantics is then extended to a larger subset, including the not and var primitives. Our construction is given as least fixpoint of an immediate consequences operator. This leads to simple definitions of abstractions useful for program analysis, which turn out to be more accurate than the usual abstractions of the pure logic programs semantics. Some new problems arise in the abstraction process. The most interesting one is the need of consistency approximations of constraints. We suggest a possible solution to these problems. Finally, we present an implementation of this new abstract interpretation framework and an instance of it for groundness analysis.

1 Logic programming vs. PROLOG

Our goal here is program analysis through abstract interpretation [7]. Hence we need a semantics for PROLOG designed so as to make easy the abstraction process and which describes the PROLOG behavior, also in its more "operational" aspects, like the cut (!) operator. All the semantics for PROLOG proposed so far do not satisfy at least one of the above requirements. Either they are easy to abstract, yet too far from the real PROLOG behavior, or they are too complex for the abstraction process but very close to the operational behavior of PROLOG programs.

One interesting recent contribution, oriented towards abstract interpretation, is the framework described in [12], which handles PROLOG with cut and shows that the analysis precision can indeed be significantly improved by taking into account the meaning of cut. The semantics is given in a goal–dependent denotational style.

Our aim is to develop a framework for goal–independent analysis, based on a bottom-up semantics. Several T_P-based bottom-up semantics have been defined for logic programs. We base our work on the s–semantics [8], which models computed answer substitutions in a goal–independent fashion but is still computable in a top–down way. The s–semantics can be easily abstracted, as shown in [3, 11, 5]. The first attempt at modelling within the s–semantics approach the depth–first search rule

of PROLOG was that of Bossi et al. [4]. The usual computed answers semantics is obtained from the more concrete semantics of resultants [10], by taking the heads of the "closed" resultants of the form $A : -$ and by discarding all the other resultants. The idea in [4] was that of abstracting a non-closed resultant of the form $A : -B$ to the "divergent" atom $\widetilde{A}$. A divergent atom $\widetilde{A}$ means that the computation related to the atom A is incomplete and thus potentially divergent. Some technical problems of the semantics in [4] have been solved in [13], where the semantics is developed for a CLP version of the original PROLOG programs. Constraints are equalities between terms and are possibly labeled by a divergence token. A potentially divergent constraint, if consistent, does not allow to observe any constraint that follows it. The concepts of "following" and "preceding" are related to the choice of sequences rather than sets of constraints as semantic domain. In [13] the same semantics exploits as semantic domain sets of constraints with an associated computation history token. Clearly the sequence structure can be obtained as an abstraction of the sets of constraints with history, but histories are useful if we have to describe other control aspects of PROLOG, as we will do in this paper.

The semantics of [13] was the starting point of our work. Our goal was extending that approach to develop a semantics able to capture more "control" features and built-in's and of showing how this semantics can be abstracted. This paper is a reduced version of [15], where all the proofs and more details can be found.

We recall that a sequence is an ordered collection of elements with repetitions; a binary string is a sequence of 0's and 1's. The concatenation of two binary strings is denoted by ::. The empty string is denoted by ϵ and is the identity with respect to ::. The prefix order on the set of all binary strings is defined as $s_1 \leq_{pf} s_2$ if and only if $s_2 \equiv s_1 :: h$ for some binary string h. The lexicographic order on binary strings is defined too. For example $1 \leq_{lex} 11$, $101 \leq_{lex} 110$. Note that $s_1 \leq_{pf} s_2$ entails $s_1 \leq_{lex} s_2$ and that $\leq_{lex}$ is a total order on the set of all binary strings.

2 Fixpoint semantics of PROLOG

We shall now develop an extension of the s-semantics able to model the depth-first search strategy and the cut built-in. This semantics can be then extended towards other operational aspects of PROLOG using the classical "equivalences" between built-in's.

Modelling the depth-first search strategy means discarding all the computed answer constraints to the right of the first infinite path of the SLD tree. Modelling the cut predicate means discarding all the computed answer constraints cut by a computed answer constraint to its left which has not been cut.

In [13] potentially divergent computations are modelled through divergent constraints. The observable function is defined as the set of convergent constraints not preceded within the sequence by a consistent divergent constraint. Constraint sequences are essentially the abstraction of the frontiers of SLD trees. Divergent constraints are abstractions of those paths in the tree which correspond to partial (i.e., not yet terminated) computations.

The cut raises some new problems because it has the twofold function of cut within a clause and of cut between clauses, it has a scope and, finally, the computation goes "through" the cut, and thus its behaviour is very different from that of divergent computations. Because of the last point, the denotation must contain some "cut conditions" together with potentially cut answer constraints. This problem does not arise in the characterization of divergence, and its solution is a key-point towards the extension of the s-semantics to full PROLOG. In terms of abstract interpretation, we cannot abstract the SLD tree only to its frontier, but we must keep also some internal details of the computation (of the tree).

2.1 The formalization

Definition 1 *A simple constraint is an element of the set $\mathcal{C}_S = \mathcal{C} \cup \widetilde{\mathcal{C}} \cup \overline{\mathcal{C}}$, where $\widetilde{\mathcal{C}} = \{\widetilde{c} | c \in \mathcal{C}\}$ is the set of divergent constraints and $\overline{\mathcal{C}} = \{\overline{c} | c \in \mathcal{C}\}$ is the set of cut internal constraints. The modulus of a simple constraint is $|s| = c \in \mathcal{C}$, where $s = c$ or $s = \widetilde{c}$ or $s = \overline{c}$. The operations of conjunction and cylindrification on $\mathcal{C}_S$ are defined for all $s_1, s_2, s \in \mathcal{C}_S$ and $c \in \mathcal{C}$ as follows:*

- $s_1 \wedge s_2 = s_1$ *if* $s_1 \in \widetilde{\mathcal{C}} \cup \overline{\mathcal{C}}$,
- $s \wedge \overline{c} = \overline{s \wedge c}$ *if* $s \in \mathcal{C}$,
- $s \wedge \widetilde{c} = \widetilde{s \wedge c}$ *if* $s \in \mathcal{C}$,
- $\exists_x \overline{c} = \overline{\exists_x c}$,
- $\exists_x \widetilde{c} = \widetilde{\exists_x c}$.

The constraints in $\overline{\mathcal{C}}$ are those that preserve some internal details of the computation, as already mentioned.

Definition 2 *A node is a triple $n = \langle s, h_1, h_2 \rangle$, where $s \in \mathcal{C}_S$, $h_1 \in \{0,1\}^*$ and $h_2 \in \{0,1\}^* \cup \{\flat\}$. The set of all the nodes is denoted by $Node$. If $n = \langle s, h_1, h_2 \rangle \in Node$, then $c(n) = s$, $His(n) = h_1$ and $cp(n) = h_2$. $c(n)$, $His(n)$ and $cp(n)$ are the constraint of n, the history of n and the cut point of n, respectively. The orders $\leq_{lex}$ and $\leq_{pf}$ are extended to the cut point set by defining $\flat \leq_{lex} \epsilon$ and $\flat \leq_{pf} \epsilon$ and by making them the reflexive and transitive closure of the resulting relations. We define $h :: \flat = \flat$ for all histories h or for $h = \flat$. We require the integrity constraints $h_2 \leq_{pf} h_1$ and, if $c(n) \in \widetilde{\mathcal{C}}$, then $cp(n) = \flat$ to be satisfied.*

$\leq_{lex}$ and $\leq_{pf}$ are two partial orders on the set of histories (also extended with $\flat$). Intuitively the cut point means that, if the constraint of the node is consistent, all the nodes following that node should be discarded, provided that they are obtained through a root-node path passing through a node with history equal to the cut point. Here the history of a node represents the path leading from the root to the node. The cut point means that if the constraint of the node is consistent, the set of nodes of the subtree rooted in the node with history equal to the cut point must be discarded,

provided they are to the right of the cutting node. Note that this node belongs to the subtree because its cut point is by definition a prefix of its history.

The integrity constraint that requires $\mathfrak{d}$ to be the cut point related to a divergent constraint is justified by the observation that such a node cuts all the nodes following it to its right.

Thus each node is potentially a cutting one. In the worst case it cuts an empty set of nodes, when its history and its cut point are equal.

The cut point $\mathfrak{d}$ represents a still undefined scope. The meaning is that the scope of the cut is currently the complete tree (as if the cut point were ϵ). However it can later be extended to the union of the sequences describing the semantics of two clauses for the same procedure (see the $\oplus$ operation and the rule $h :: \mathfrak{d} = \mathfrak{d}$), or it can be bounded to the tree containing the node, through the operator μ (see later).

Definition 3 *We define on Node the following monotonic operations:*

- $\exists_x \langle s, h_1, h_2 \rangle = \langle \exists_x s, h_1, h_2 \rangle$ *for every variable* x,
- $\langle s_1, h_1', h_2' \rangle \otimes \langle s_2, h_2', h_2'' \rangle = \langle s_1, h_1', h_2' \rangle$ *if* $s_1 \in \widetilde{\mathcal{C}} \cup \overline{\mathcal{C}}$,
- $\langle s_1, h_1', h_2' \rangle \otimes \langle s_2, h_2', h_2'' \rangle = \langle s_1 \wedge s_2, h_1' :: h_2', h_1' :: h_2'' \rangle$ *if* $s_1 \in \mathcal{C}$.

Our semantic domain consists of sets of nodes that can be thought of as the abstraction of the frontier of an SLD tree together with some internal nodes of the tree, abstracted by the nodes with constraint in $\overline{\mathcal{C}}$. Thus two nodes in a sequence cannot have the same history if their constraints are not in $\overline{\mathcal{C}}$ and for every history there must exist a node belonging to the sequence with constraint not in $\overline{\mathcal{C}}$ and having as history a prefix of such a history, or vice versa.

Definition 4 $C_f \subset \mathcal{P}_f(Node)$ *is defined as follows.* $F \in C_f$ *if and only if*

1. $n_1, n_2 \in F$, $c(n_1), c(n_2) \notin \overline{\mathcal{C}}$ *and* $His(n_1) \leq_{pf} His(n_2)$ *entail* $n_1 = n_2$,
2. *for every history* h *there exists* $n \in F$ *such that* $c(n) \notin \overline{\mathcal{C}}$ *and moreover either* $His(n) \leq_{pf} h$ *or* $h \leq_{pf} His(n)$.

Definition 5 *Assume* $\langle s, h_1, h_2 \rangle \in Node$, $h :: \langle s, h_1, h_2 \rangle = \langle s, h :: h_1, h :: h_2 \rangle$ *and* $F_1, F_2 \in C_f$.
$\exists_x F_1 = \{\exists_x n | n \in F_1\}$ *and* $F_1 \oplus F_2 = \{0 :: n | n \in F_1\} \cup \{1 :: n | n \in F_2\}$.

The definition of the multiplication on sequences is more complex. When we expand the frontier of the SLD tree we must keep a copy of the cutting nodes through cut internal nodes.

Definition 6 *The map* $\otimes : C_f \times C_f \longrightarrow C_f$ *is defined as follows. For all* $F_1, F_2 \in C_f$,

$$F_1 \otimes F_2 = \{n_1 \otimes n_2 | n_1 \in F_1, n_2 \in F_2\} \cup \{\langle \overline{c}, h_1, h_2 \rangle | \langle c, h_1, h_2 \rangle \in F_1, h_2 <_{pf} h_1\}.$$

In order to understand the meaning of the above definition let us consider the following PROLOG program.

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

which, when represented by our CLP-like syntax, becomes

```
p(x):-(cut(q(x)) and r(x)) or x=3.
q(x):-x=1 or x=2.
r(x):-x=2.
```

The "semantics" of `q(`x`)` and of `r(`x`)` are straightforward.

$$S_{q(x)} = \{\langle x = 1, 0, 0 \rangle, \langle x = 2, 1, 1 \rangle\}$$

$$S_{r(x)} = \{\langle x = 2, \epsilon, \epsilon \rangle\}$$

Which is the semantics of `cut(q(`x`))`? We will see later that it is

$$S_{cut(q(x))} = \{\langle x = 1, 0, \mathfrak{d} \rangle, \langle x = 2, 1, \mathfrak{d} \rangle\},$$

i.e. every node cuts all the nodes which follow it. The semantics of `cut(q(`x`))` and `r(`x`)` will then be $S_{cut(q(x))} \otimes S_{r(x)}$, that is

$$S_{cut(q(x)) \text{ and } r(x)} = \{\langle \overline{x = 1}, 0, \mathfrak{d} \rangle, \langle false, 0, 0 \rangle, \langle \overline{x = 2}, 1, \mathfrak{d} \rangle, \langle x = 2, 1, 1 \rangle\}.$$

The nodes with constraint in $\overline{\mathcal{C}}$ have been originated from the nodes in $S_{cut(q(x))}$ with convergent constraint and with cut point strictly less than their history. They are nodes which are "really" cutting (i.e. that cut a non empty set of nodes). Their constraints must be kept because they are needed for the computation of the observable function. From the last sequence we conclude that the computed answer constraint $x = 2$ is not observable because it is cut. The naive definition of the $\otimes$ operation, i.e. the usual cartesian product of constraints, is instead erroneous, because it would lead to the wrong conclusion that

$$S'_{cut(q(x)) \text{ and } r(x)} = \{\langle false, 0, 0 \rangle, \langle x = 2, 1, 1 \rangle\}.$$

Consider now the last step. The semantics of `p(`x`)` is $S_{cut(q(x)) \text{ and } r(x)} \oplus S_{x=3}$, that is

$$S_{p(x)} = \{\langle \overline{x = 1}, 0, \mathfrak{d} \rangle, \langle false, 0, 0 \rangle, \langle \overline{x = 2}, 1, \mathfrak{d} \rangle, \langle x = 2, 1, 1 \rangle\} \oplus \{\langle x = 3, \epsilon, \epsilon \rangle\}$$
$$= \{\langle x = 1, 00, \mathfrak{d} \rangle, \langle false, 00, 00 \rangle, \langle \overline{x = 2}, 01, \mathfrak{d} \rangle, \langle x = 2, 01, 01 \rangle, \langle x = 3, 1, 1 \rangle\}.$$

Note that the cut point of the first node is still $\flat$, because $0 ::\flat = \flat$. This choice has been very clever. If we confuse $\flat$ with ϵ we obtain $0 :: \epsilon = 0$, which does not allow the first node to cut all the following nodes. The "open" scope will eventually be "closed" by the map μ (see later).

We need a function to "execute a cut". All the nodes of the frontier of the SLD tree (thus not the nodes with constraint in $\overline{C}$), take $\flat$ as cut point, thus cutting all the nodes which follow them to the right.

Definition 7 *The map* $! : Node \longrightarrow Node$ *is defined as follows.*

$$!(\langle s, h_1, h_2\rangle) = \begin{cases} \langle s, h_1, h_2\rangle & \text{if } s \in \overline{C} \\ \langle s, h_1, \flat\rangle & \text{otherwise} \end{cases}$$

and is extended to C_f*, by defining* $! : C_f \longrightarrow C_f$ *such that, for all* $F \in C_f$*:* $!(F) = \{!(n) | n \in F\}$.

We need also a function to "close" an "open" scope. This will be done at every choice point.

Definition 8 *The map* $\mu : Node \longrightarrow Node$ *is defined as follows.*

$$\mu(\langle s, h_1, h_2\rangle) = \begin{cases} \langle s, h_1, \epsilon\rangle & \text{if } h_2 = \flat \text{ and } s \notin \overline{C} \\ \langle s, h_1, h_2\rangle & \text{otherwise} \end{cases}$$

and is extended to C_f*, by defining* $\mu : C_f \longrightarrow C_f$ *such that, for all* $F \in C_f$*:* $\mu(F) = \{\mu(n) | n \in F\}$.

Note that the classical "cut idempotence" folk theorem becomes a toy problem in our framework. We just need to note the idempotence of the ! map.

Definition 9 *An environment is a function which maps every function symbol to an element of* C_f*:* $Env_f = Pred \longrightarrow C_f$.

Let us show now how to compute the semantics (denotation) of an goal in an environment.

Definition 10 *The map* $\mathcal{E}[\![\_]\!]\_ : Goal \times Env_f \longrightarrow C_f$ *is defined as follows.*

$$\begin{gathered} \mathcal{E}[\![c]\!]e = [\![c]\!] \\ \mathcal{E}[\![A_1 \text{ and } A_2]\!]e = \mathcal{E}[\![A_1]\!]e \otimes \mathcal{E}[\![A_2]\!]e \\ \mathcal{E}[\![A_1 \text{ or } A_2]\!]e = \mathcal{E}[\![A_1]\!]e \oplus \mathcal{E}[\![A_2]\!]e \\ \mathcal{E}[\![\texttt{exists } x.A]\!]e = \exists_x \mathcal{E}[\![A]\!]e \\ \mathcal{E}[\![\mathtt{p}(x)]\!]e = \mu(\exists_\alpha([\![\delta_{\alpha,x}]\!] \otimes e(p))) \\ \mathcal{E}[\![\mathtt{cut}(A)]\!]e = !(\mathcal{E}[\![A]\!]e) \end{gathered}$$

Definition 11 *The immediate consequences operator* $T_P : Env_f \longrightarrow Env_f$ *related to the program* P *is defined as follows. For every predicate symbol* p *in* P,

$$T_P(e)(p) = \exists_{x_p}([\![\delta_{\alpha,x_p}]\!] \otimes \mathcal{E}[\![A_p]\!]e),$$

where A_p *is the body of the clause defining* p *in* P.

We can think of a divergent constraint as a cut constraint with scope equal to the whole portion of tree to its right. Thus the observable nodes of a sequence are those nodes with consistent constraint and not cut by a node, which in turn is not cut and has a consistent constraint. Let us define the predicate CUT.

Definition 12 $CUT(\langle s_1, h_1, cp_1\rangle, \langle s_2, h_2, cp_2\rangle)$ *is true if and only if* $cp_1 \leq_{pf} h_2$, $h_1 <_{lex} h_2$, $h_1 \not\leq_{pf} h_2$, $|s_1| \neq false$.

The observable set $O : C_f \longrightarrow \mathcal{P}_f(C)$ is recursively defined as follows.

Definition 13 *For all* $F \in C_f$*:* $O(F) = \{n \in B | c(n) \in C\}$, *where* $B = \{n \in F || c(n)| \neq false, \neg\exists n' \in B$ *such that* $CUT(n', n)\}$.

It can be shown that the recursive equation defining B has exactly one solution. Moreover $O(F)$ can be computed through a simple algorithm (see [15]).

We finally define the denotation of a goal in a program.

Definition 14 *Assume* $A \in Goal$ *and* P *be a program. The semantics of* A *in* P *is* $D(A, P) = O(\mathcal{E}[\![A]\!](lfp(T_P)))$.

Note that the resulting semantics is AND–compositional by definition.

By extending T_P, $\mathcal{E}[\![A]\!]$ and O to continuous maps on complete lattices we can conclude that

Proposition 15 $D(A, P) = \bigcup_{i \geq 0} O(\mathcal{E}[\![A]\!]T_P^i(\bot))$.

In [15] a more concrete semantics is defined, from which one can obtain the preceding one by means of abstraction. This new semantics is given as an unfolding semantics and is proved to be correct and complete with respect to the computed answer constraints of PROLOG with cut. A tree of the unfolding semantics is abstracted in this way: the leaves are abstracted into convergent or divergent constraints, while some internal nodes are abstracted into cut internal constraints.

3 Abstract semantics

The abstraction of our fixpoint semantics is similar to that of the semantics described in [13]. We need an approximation of the constraints and an approximation of the sequences of constraints. Constraint approximation can be done as in the pure logic programming paradigm. Unfortunately the depth–first search rule and the cut introduce some "non-monotonic" behaviours in the abstract observable function and force us to use downward and upward approximations of the consistency of a constraint (control approximation), together with the classical abstraction of a constraint in the domain of analysis (logic approximation). Roughly speaking, downward and upward approximations of the consistency of a constraint are needed to determine when a constraint is consistent and then cuts a set of constraints in its scope; we need an approximation in both direction to avoid wrong conclusions on the set of observable constraints.

As consistency approximation is totally unrelated to logic approximation, if we develop a general way for doing downward and upward approximation of consistency, then we get a general framework for the abstract analysis of Prolog programs, parametric w.r.t. the domain of analysis.

In [15] we developed an abstract semantics where every constraint is abstracted in three ways (downward and upward consistency approximation and domain dependent approximation); all other details of the semantics are similar to those of the preceding concrete semantics; the abstract domain is formed by sequences of abstract constraints. The local and then global correctness of this abstraction can be showed, following the approach of the Galois connections. We showed also as the upward and downward approximations of the consistency of the constraints can be used to obtain a superset of the set of constraints observable from the Prolog viewpoint, through a simple algorithm.

3.1 Control approximation

The key problem is now consistency approximation, i.e. control approximation. If we show how it can be done we will construct an abstraction framework parametric with respect to the kind of analysis we are doing and we will be able to instantiate our framework using the classical domains developed for the pure logic programming paradigm.

The problem of control approximation was already tackled in [2], leading to the need for a "termination theory" for PROLOG, which was however not defined. The only approach we know of which solves a similar problem is that presented in [1]. It is a technique to deal with upward approximations of the consistency of sets of equations in the equational logic programming framework. This technique is useful for our aims. The approximation in [1] is performed by replacing the subterms, occurring at a depth greater than a given k, by fresh existentially quantified variables. This approximation is similar to the classical depth–k abstraction.

How does this apply to downward approximations? Now the solution is easy. We can just replace in a term its subterms, occurring at a depth greater than a given k, by new distinct constants. In this way we obtain a constraint "less consistent" than the original one.

We will denote the new constants or variables introduced in the abstraction process by indexed stars.

This abstraction process hides some quibbles related to the occur–check. For example, the concrete constraint $\{x = f(x)\}$ is not consistent in the finite trees domain, while its approximation with $k = 1$, that is $\{x = f(\star_1)\}$, is satisfiable in that domain. Obviously we have lost the information that $\star_1$ is a term that contains the variable x. The most simple solution to this problem is that of interpreting PROLOG on the rational trees domain [6].

This is formally our abstraction scheme:

Definition 16 *Let E be the set of equations $E = \{\exists X_1 s_1 = t_1, \ldots, \exists X_n s_n = t_n\}$, and assume $k \geq 1$.*

$$\zeta(E,k) = \{\exists X_1' \zeta(s_1,k,1) = \zeta(t_1,k,1), \ldots, \exists X_n' \zeta(s_n,k,1) = \zeta(t_n,k,1)\},$$

where X_i' is obtained from X_i by discarding the variables not present neither in $\zeta(s_i,k,1)$ nor in $\zeta(t_i,k,1)$ and by adding the $\star$–variables introduced by the function ζ. Moreover,

- $\zeta(x,k,h) = x$ *if x is a variable and $h \leq k$,*
- $\zeta(f(t_1,\ldots,t_n),k,h) = f(\zeta(t_1,k,h+1),\ldots,\zeta(t_n,k,h+1))$ *if $h \leq k$,*
- $\zeta(t,h) = \star_j$ *if $h > k$ and $\star_j$ is a fresh existentially quantified variable.*

Dually we define:

$$\beta(E,k) = \{\exists X_1' \beta(s_1,k,1) = \beta(t_1,k,1), \ldots, \exists X_n' \beta(s_n,k,1) = \beta(t_n,k,1)\},$$

where X_i' is obtained from X_i by discarding the variables not occurring neither in $\beta(s_i,k,1)$ nor in $\beta(t_i,k,1)$. Moreover,

- $\beta(x,k,h) = x$, *if x is a variable and $h \leq k$,*
- $\beta(f(t_1,\ldots,t_n),k,h) = f(\beta(t_1,k,h+1),\ldots,\beta(t_n,k,h+1))$ *if $h \leq k$,*
- $\beta(t,h) = \star_j$ *if $h > k$ and $\star_j$ is a new constant.*

Definition 17 *The conjunction, cylindrification and unification operations are defined as follows.*

- $E_1 \dot{\wedge} E_2 = E_1 \cup E_2'$, *where E_2' is obtained from E_2 by renaming apart the $\star$–variables,*
- $\dot{\exists}_x \{\exists X_1 s_1 = t_1, \ldots, \exists X_n s_n = t_n\} = \{\exists X_1' s_1 = t_1, \ldots, \exists X_n' s_n = t_n\}$, *where $X_i' = X_i \cup \{x\}$,*
- $\dot{\delta}_{x,y} = \{x = y\}$

Downward definitions are dual.

In order to guarantee that our abstract domain is noetherian, we have to perform a simplification process on constraints, by transforming them in solved form and then projecting them on the variables of the original goal. Since the number of variables in a goal is bounded, our constraint domain is noetherian. But is our analysis finite? clearly, being our abstract domain made of abstract constraint sequences, the abstract domain isn't noetherian even if the abstract constraint domain is. What one can do is to keep a portion of a sequence and to "abstract" the remaining portion. An alternative approach is suggested in [13]: the key idea is discard repeated occurrences of the same constraint in a sequence. If the constraint domain is noetherian, as is our case, the computation must terminate even without sequence approximation [9]. Unfortunately,

this is no longer possible if we take the cut into account, since discarding repeated constraints is no more a safe operation, due to the cut scope. However, in [14] some safe and loss-free simplification rules have been developed and proved correct; though they work properly on a big subset of Prolog programs, we still lack the characterization of the class of Prolog programs for which our analysis is finitely computable. The description of these simplification rules is out of the scope of this paper; moreover this topic is still under development.

4 An implementation

An implementation of our abstract interpretation framework has been carried over in order to show its feasibility and to collect suggestions about how push our research efforts toward a more useful and simpler abstract interpretation framework. Our implementation has been developed using the SWI-Prolog compiler.

The analyzer accepts as input a Prolog program and some user queries and produces some information about the abstract execution of them in the program. Going a little deeper, the analyzer can be split in four portions: the first one computes the abstract fixpoint of the program, the second one evaluates the user queries in this abstract fixpoint; the third and fourth portions are devoted to the domains for control and logic approximation, respectively. Control approximation is done using downward and upward approximation of the consistency of the constraints, while logic approximation is left unspecified; the programmer can instantiate the logic component of the analyzer with whatever abstract domain he wants, provided he implements some operation on this domain, like meet, restriction or equivalence test; all control information has been implemented once and for all in the control portion of the analyzer, so the programmer won't worry about it. Moreover the programmer doesn't need to implement a join operator, because non-determinism isn't abstracted but is maintained in the sequence structure; this is often a big benefit.

4.1 An example

We present now an example of abstract analysis carried over with our abstract analyzer; we instantiated it with the classical domain for groundness analysis with aliasing, known as $\mathcal{DEF}$; we choosed $\mathcal{DEF}$ rather than $\mathcal{POS}$ because we don't need a disjunction operator on constraints; this is an example of the benefit obtained because we don't specify a join operator; clearly the overhead has been shifted in the abstract analyzer that need to deal with abstract sequences rather than simpler sets.

Let's consider the following program, contained in the file **example**:

```
norm(X,XN):-var(X),!,transform(X,XN).
norm(X,XN):-extract(X,XN).

extract([],[]).
extract([s(_X)|T],[1|T1]):-extract(T,T1).
extract([0|T],[0|T1]):-extract(T,T1).

transform([],[]).
transform([s(0)|T],[1|T1]):-transform(T,T1).
transform([0|T],[0|T1]):-transform(T,T1).
```

Calling the analyzer with `go(example,2)` we get the fixpoint. The evaluation of the following query in the fixpoint shows that we get a more precise result about the exit mode of the query then an analysis for pure logic programs:

```
Query to evaluate (q to quit):
|: norm(X,[0,1,0,0,1,1,0]).

This is the denotation of your query:
query(X) :- X.
query(X) :- true.
query(X) :- X.

This is the same denotation with control approximation:
query(X) :- X, uncertain.

The execution of the query norm(X, [0,1,0,0,1,1,0])
results in the following: norm(g, [0,1,0,0,1,1,0])
```

If we take as denotation of the query the one without control approximation, when we do the lub between the three constrained atoms all we get is *true*, that is no information; our analysis concludes instead that the variable X is surely made ground.

5 Conclusions and future trends

Our fixpoint semantics is, up to our knowledge, the first goal-independent semantic characterization of PROLOG with cut oriented towards abstraction. The new problems, like consinstency approximation of constraints, have been solved aiming at a domain independent construction, so that we're able to "reuse" all abstract domains developed for the pure logic programming paradigm. An implementation has showed the feasibility of our approach.

Our results can be improved with new and more precise approximations of the consistency of the constraints. Moreover it would be useful to characterize the class of Prolog programs for which our analysis is a finite computable one.

References

[1] M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatisfiability for Equational Logic Programming. *Journal of Logic Programming*, 22(3):221–252, 1995.

[2] R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog Control. *Journal of Logic and Computation*, 3:579–603, 1993.

[3] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[4] A. Bossi, M. Bugliesi, and M. Fabris. Fixpoint semantics for PROLOG. In D. S. Warren, editor, *Proc. Tenth Int'l Conf. on Logic Programming*, pages 374–389, Cambridge, Mass., 1993. The MIT Press.

[5] M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. *Theoretical Computer Science*, 124(1):93–125, 1994.

[6] A. Colmerauer. Prolog and Infinite Trees. In K. L. Clark and S. A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, New York, 1982.

[7] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[8] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[9] M. Gabbrielli, R. Giacobazzi, and D. Montesi. Modular logic programs over finite domains. In D. Saccà, editor, *Proc. Eight Italian Conference on Logic Programming*, pages 663–678, 1993.

[10] M. Gabbrielli, G. Levi, and M. C. Meo. Resultants semantics for PROLOG. *Journal of Logic and Computation*, 1995.

[11] R. Kemp and G. Ringwood. An Algebraic Framework for the Abstract Interpretation of Logic Programs. In S. K. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 506–520. The MIT Press, Cambridge, Mass., 1990.

[12] B. Le Charlier, Rossi S., and P. Van Hentenryck. An Abstract Interpretation Framework which Accurately Handles PROLOG Search Rule and the Cut. In M. Bruynooghe, editor, *Proceedings of the 1994 Int'l Symposium on Logic Programming*, pages 157–171. The MIT Press, Cambridge, Mass., 1994.

[13] G. Levi and D. Micciancio. Analysis of pure PROLOG programs. In M.I. Sessa, editor, *Proceedings GULP-PRODE '95, 1995 Joint Conference on Declarative Programming*, 1995.

[14] N. F. Spoto. Un Interprete Astratto per Programmi Logici con Cut. Developed on behalf of the Department of Computer Science of the University of Pisa; available on request e-mailing at: spoto@di.unipi.it.

[15] N. F. Spoto. Semantica Concreta e Astratta di PROLOG con Cut. Master's thesis, Dipartimento di Informatica, Università di Pisa, July 1995. In Italian.