Static Analysis of Prolog with Cut

Gilberto Filé, Sabina Rossi

Dipartimento di Matematica Pura ed Applicata
Università di Padova
Via Belzoni, 7 I-35131 Padova (Italy)
E-mail: {gilberto,sabina}@zenone.unipd.it

Abstract. This paper presents a general approach to the Abstract Interpretation of Prolog programs with cut. In most of previous approaches the cut primitive is merely ignored.
Our method consists in trasforming an interpreter for Prolog into an interpreter that computes on abstract values and that performs loop-checks by storing all encountered call patterns into a table. In order to guarantee correctness, this tabled interpreter needs information about the sure success of the corresponding concrete computations. Such information, called *control information*, is used to control the execution of the cuts by the tabled interpreter: a cut is executed only if the control information guarantees that it is also executed at the concrete level, otherwise, the cut is ignored. Control information can be easily added to any abstract domain.

Introduction

Abstract Interpretation has been successfully developed in recent years for the static analysis of programs. It has been applied to many types of languages.
Recently Abstract Interpretation techniques have been applied to logic programming and Prolog. However, little attention has been devoted to the Abstract Interpretation of "full" Prolog with cut and built-ins, except for [2, 11]. It is easy to see that ignoring the cut has a negative consequence on the quality of the information produced (not executing a cut means to consider computations that are pruned at the concrete level).
The cut is a non-logical operator whose meaning is specified only by means of an operational semantics [9].

The intuition at the basis of the Abstract Interpretation method is that the data-flow analysis of a program consists of executing it on a special domain called *abstract* because it abstracts some interesting properties of the normal concrete domain. According to this idea, the operational approach to Abstract Interpretation that we have adopted, is as follows. Let L be a programming language.

1. Using the usual approximation relation that exists between an abstract domain D and the concrete domain C of the prgrams of L, it is easy to define for each concrete program P of L a corresponding abstract program P'.

2. Let I be an intepreter of L and use I to interpret P': such execution is a data-flow analysis of the execution of P by I.
3. Even if D is a finite set, the execution of P' by I can be not finite. In order to guarantee termination on finite domains, a loop-check mechanism is added to I.

This operational approach has been adopted in [3] for logic and constraint programs and in [10] for logic programs. In these works the interpreter used was simply a program (ND-I) that nondeterministically traverses the LD-trees. ND-I was sufficient because no control built-in like cut was considered, and thus, it was not necessary to model the depth-first left-to-right traversal of a standard Prolog interpreter (called standard traversal in what follows). At the contrary, this becomes necessary in this paper where we want to treat also the cut operator. To this end we consider the standard Prolog interpreter St-I that plays the role of I in point 2 above. The program for the data-flow analysis is obtained adding a loop-check in the form of a tabulation mechanism to St-I, cf. point 3 above. This program is called St-TI. Intuitively, a tabulation mechanism consists of collecting in a table all the atoms A, called during an execution, together with the solutions found for them. In this way, when an equivalent call A' is encountered later on, the solutions of A are used to expand A'. In what follows A is called a solution call, since it produces solutions, whereas A' is called a look-up call because it just looks up the solutions of A. The addition of a tabulation mechanism to St-I gives rise to several problems that are new wrt [3, 10]. The main one is as follows. Assume that, according with the operational approach described above, the data-flow analysis of a Prolog program P, is obtained by executing the corresponding program P' with the tabled interpreter St-TI. If P has cuts, so will P' and they will be executed by St-TI just as by St-I. This can lead to incompleteness, in fact, the execution of P', being on the abstract domain D (i.e., simpler than the concrete one), can lead to more derivations than the execution of P. Hence, a cut of P' may be executed whereas the corresponding derivation of P fails before such point. If this happens the execution of the cut may prune computations that, at the contrary, are not pruned for P and thus these concrete computations are not analyzed: incompleteness! The solution to this problem is to enhance the abstract domain D with some extra information, called control information, in such a way that St-TI, when executing P' on this new abstract domain, *knows* whether the corresponding concrete computations are all successful. Only in this case a cut is executed. Control information can be added to every abstract domain.

The rest of the paper is as follows. Section 1 contains some preliminary definitions. In Section 2 the tabulation mechanism in [3] is described by means of an example. The necessity of modeling the standard traversal when treating the cut and the inappropriateness of the original mechanism in this case are all explained by means of examples. In Section 3 the extensions of the original tabulation mechanism needed for modelling the standard traversal and handling the cut are described. Finally, in Section 4 some more examples show the need of control information for ensuring completeness and explain its use.

1 Preliminaries

The reader is assumed to be familiar with the basic concepts of Logic Programming, see [1, 12].
$\mathbf{V}$ is an infinite set of variables. A substitution is a mapping from $\mathbf{V}$ to $T(\Sigma, \mathbf{V})$ which acts as the identity almost everywhere. A renaming is a bijection on variables. With $Subst$ we denote the set of all idempotent substitutions and with $\theta_{|_V}$ the restriction of θ to the set of variables V.
The list operation $tail$ is as follows: $tail(nil) = nil$ and $tail([s \mid L]) = L$.
If o is a sintactical object, then $Var(o)$ is the set of all variables occurring in o.
A *computation system* $\mathcal{D}$ is a 5-tuple, $(D, \Phi, \Pi, \leq, \equiv)$, where:

1. D is a complete lattice, called *computation domain*. Its bottom element is $\bot_D$, and for all $d \in D$ and for all renamings σ, $d\sigma \in D$; with CP_D we denote the set of all call patterns on D, i.e., the set of all pairs $[A, d]$ such that A is an atom and $d \in D$;
2. Φ is a function on D of the type: $(D^2 \times Subst) \rightarrow D$;
3. Π is a projection of the type: $(D \times \wp(\mathbf{V})) \rightarrow D$, such that, if $d \in D$ and $V \in \wp(\mathbf{V})$, then $\Pi(d, V)$ projects d onto the variables of V;
4. $\leq$ is a partial order on D;
5. $\equiv$ is an equivalence relation on CP_D such that for all $[A, d] \in CP_D$ and for all renaming substitution σ, $[A, d] \equiv [A, d]\sigma$.

$\mathcal{D}$ is called *finite* if its computation domain D is finite modulo renaming.
The function Φ represents the unification function on D.
A *normalized atom* is an atom of the form $p(\bar{X})$ containing distinct variables.
A *normalized program* P over $\mathcal{D}$ is a finite sequence of definite clauses of the form: H:$-d, A_1, \ldots, A_k$, where $H, A_1, \ldots, A_k$ are either normalized atoms or the control primitive cut, and $d \in D$.
A *normalized goal* is a normalized clause whose head is empty.
In the following only normalized programs and goals are considered.
Observe that considering normalized programs brings no loss of generality: any Prolog program can be put into this form where d is a substitution.
The usual *concrete computation system* of Prolog is

$$\mathcal{C}_s = (Subst, \Phi_s, \Pi_s, \subseteq, \equiv_s)$$

where Φ_s is the unification function on $Subst$; Π_s is defined by: $\Pi_s(\theta, V) = \theta_{|_V}$; $\equiv_s$ is the equivalence relation on CP_{Subst} defined by: $[A_1, \theta_1] \equiv_s [A_2, \theta_2]$ iff there exists a renaming σ such that $(A_1\theta_1)\sigma = A_2\theta_2$.
As an example, `append` over $\mathcal{C}_s$ is:

`append`(X, Y, Z):$-\{X/[S \mid T], Z/[S \mid U]\}$, `append`$(T, Y, U)$.
`append`(X, Y, Z):$-\{X/[\,], Y/Z\}$.

Let P be a program with goal G_0. The LD-*tree* of $P \cup \{G_0\}$ is the SLD-tree [12] corresponding to the Prolog selection rule.
Consider any LD-tree and a node G_i in it containing a cut. The *parent node* of

this cut is the ancestor of G whose expansion has produced this cut.
The *Standard Interpreter, St-I*, is a program that explores the LD-trees by employing the *standard traversal* rule and by executing cuts according to their operational semantics, i.e., pruning all the branches to the right of a path from the parent goal to the goal that has executed the cut.
We recall below the notion of abstraction between computation systems.
Let $\mathcal{D} = (D, \Phi_D, \Pi_D, \leq_D, \equiv_D)$ and $\mathcal{C} = (C, \Phi_C, \Pi_C, \leq_C, \equiv_C)$ be two computation systems. $\mathcal{D}$ *abstracts* $\mathcal{C}$ iff the following properties hold:

1. there exists a function γ, called *concretization function*, from D to 2^C such that γ is monotone;
2. $\forall \delta \in Subst$, $\forall c_1, c_2 \in C$ and $\forall d_1, d_2 \in D$, $c_1 \leq_C \gamma(d_1), c_2 \leq_C \gamma(d_2)$ implies $\Phi_C(c_1, c_2, \delta) \leq_C \gamma(\Phi_D(d_1, d_2, \delta))$;
3. $\forall V \in \wp(\mathbf{V})$, $\forall c \in C$ and $\forall d \in D$, $c \leq_C \gamma(d)$ implies $\Pi_C(c, V) \leq_C \gamma(\Pi_D(d, V))$.

Given a program P over $\mathcal{C}$, a corresponding program P' over $\mathcal{D}$ is as follows: for each clause $H{:-}c, A_1, \ldots, A_k$ of P P' has a clause $H{:-}d, A_1, \ldots, A_k$ such that $c \leq_C \gamma(d)$.

2 Tabled Computations for Pure Prolog Programs

In this section we describe, by means of examples, the tabled interpreter $ND\text{-}TI$ presented in [3]. Later it will be shown that $ND\text{-}TI$ is unsuitable if one wants to treat the cut operator.
Let $\mathcal{D}$ be an arbitrary computation system. In this section pure Prolog programs (i.e. without cuts) over $\mathcal{D}$ are considered.
Given a goal $G = {:-}d, A_1, \ldots, A_k$, the *leftmost call pattern* of G, noted $lf(G)$, is the call pattern $[A_1, \Pi(d, Var(A_1))]$ over $\mathcal{D}$.
For example, consider the goal over $\mathcal{C}_s$:

$$G = {:-}\{X = [a|U], T = [a|W]\}\texttt{append}(X, Y, Z), \texttt{append}(T, Z, V).$$

Its leftmost call pattern is: $lf(G) = [\texttt{append}(X, Y, Z), \{X = [a|U]\}]$.
The idea of $ND\text{-}TI$ is as follows: collect in a table all the leftmost call patterns of the goals found so far in the computation with their computed solutions, and whenever a new goal G is produced, check whether the table already contains a call pattern $lf(G')$ equivalent to $lf(G)$. In this case use the solutions of $lf(G')$, collected in the table, for expanding G.
$ND\text{-}TI$ constructs a *tree-table* pair (t, T), as follows. T is a table where each entry consists of a *key*, which is a call pattern, and a *solution list* associated to that key, which is a list of distinct elements of the computation domain D. t is similar to an LD-tree. It contains two types of nodes: nodes that generate an entry in the table, called *solution* nodes and nodes that use these solutions, called *look-up* nodes.
The keys of the table are call patterns $[A, d]$, where $Var(d) \subseteq Var(A)$. Also in the tree it is convenient to keep values d projected. Every goal in the tree will be ${:-}d, A_1, \ldots, A_k$ where $Var(d)$ is included in the variables of the clause that

has A_1 in its body. In order to manage correctly the switches of sets of variables during the computation, it is necessary to introduce special values called *call-exit markers*. A call-exit marker contains the information $e = [cs_1, cs_2, A = H, d]$. Suppose to have a goal $G = :-d, A, R$, where A is an atom of the body of cs_1 such that A is unifiable with the head of cs_2. Then the new goal will be $:-d'', B_1, \ldots, B_m, [cs_1, cs_2, A = H, d], R$, where $d'' = \Pi(\Phi(d, d', \delta), Var(cs_2))$, $\delta = mgu(A = H)$ and $cs_2 = H:-d', B_1, \ldots, B_m$. The call-exit marker e is said to correspond to $lf(G) = [A, \bar{d}]$, where $\bar{d} = \Pi(d, Var(A))$.
When e becomes the leftmost element of a goal, it means that a refutation for A under d has been computed and a solution for it has been reached.
Call-exit markers are useful in order to know when new solutions for a left-most call pattern must be added to the table.

Example 1. Let P be the program:
$q(X):-d, p(X)$; $r(X):-d, p(X)$;
$p(X):-d'$; $p(X):-d''$.
and G_0 be the goal $:-d_0, q(X), r(Y)$, where $d = d_0 = \varepsilon$, $d' = \{X/a\}$ and $d'' = \{X/b\}$.
By executing P with G_0 by *ND-TI*, one obtains the tree-table pair represented in Fig. 1. At the beginning the table is empty.
Since it does not contain a key equivalent to $lf(G_0)$, a new entry with key $lf(G_0) = [q(X), d_0]$ and empty solution list is added to it. G_0 is a solution node. Selecting the left most atom of G_0 and using the renamed first clause, cs_1, $q(X_1):-d, p(X_1)$, in P, the resolvent $G_1 = :-d_1, p(X_1), \Box_1, r(Y)$ is computed, where $d_1 = \Pi(\Phi(d_0, d, \delta), Var(cs_1)) = \{X_1/X\}$, $\delta = \{X/X_1, X_1/X\}$ and $\Box_1$ is the call-exit marker $[cs_0, cs_1, q(X) = q(X_1), d_0]$. Solving in the same way G_1, a new entry with key $lf(G_1) = [p(X_1), d_1]$ and empty solution list is added to the table. G_1 is a solution node and it is resolved by using a renaming of the third clause, cs_2. The resolvent $G_2 = :-d_2, \Box_2, \Box_1, r(Y)$ with $d_2 = \{X_2/a\}$ and $\Box_2 = [cs_1, cs_2, p(X_1) = p(X_2), d_1]$ is computed.
Consider now G_2. In general, when a node G in the tree starting with a call-exit marker, has the form $:-d, [cs_1, cs_2, A = H, d'], A_2, \ldots, A_k$, then it is resolved by computing $G' = :-d'', A_2, \ldots, A_k$, where $d'' = \Pi(\Phi(d', d, \delta), Var(cs_1))$ and $\delta = mgu(A = H)$. Moreover, the solution $\Pi(d'', Var(A))$ is added to the end of the solution list corresponding to the key $[A, \Pi(d', Var(A))]$ in the table.
Solving in this way the two call-exit markers, $\Box_2$ and $\Box_1$, and using the renamed second clause cs_3, $r(Y_1):-d, p(Y_1)$, to solve $r(Y)$, one obtains the resolvents G_3, G_4 and G_5 represented in Fig. 1, where $d_3 = \{X_1/a\}$, $d_4 = \{X/a\}$, $d_5 = \{Y_1/Y\}$, and $\Box_3 = [cs_0, cs_3, r(Y) = r(Y_1), d_4]$.
Consider now G_5. Observe that there exists in the table an entry with key $lf(G_1) = [p(X_1), d_1]$ equivalent to $lf(G_5) = [p(Y_1), d_5]$. In this case the solutions of $lf(G_1)$ are used to expand $lf(G_5)$. Thus, G_5 becomes a look-up node and a pointer to the solution list $L = [\{X_1/a\}|nil]$ of $lf(G_1)$ is added to G_5. The solution $\{X_1/a\}$ is used to expand G_5 by computing $G_6 = :-d_6, \Box_3$ where $d_6 = \Pi(\Phi(d_5, \{X_1/a\}, \delta'''), Var(cs_3)) = \{Y_1/a\}$ and $\delta''' = mgu(p(X_1) = p(Y_1))$. Then the pointer of $lf(G_5)$ is moved to $tail(L)$.

Since this pointer is now to an empty list in the table, when the computation backtracks to G_5, the expansion of G_5 is suspended. It can be continued only if a new solution for $lf(G_1)$ is added in the table. □

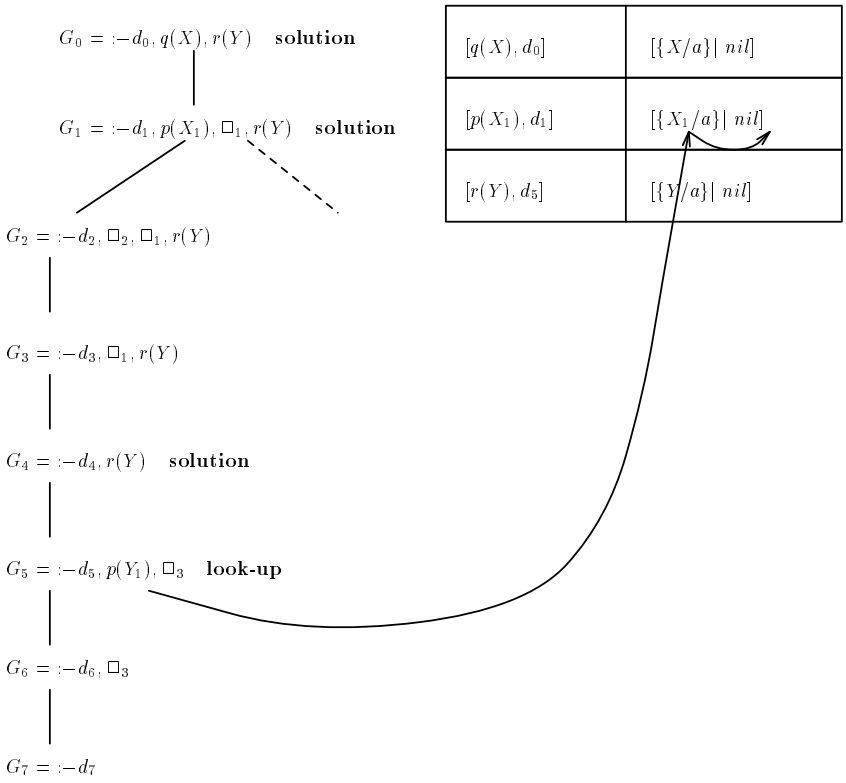


Fig. 1.

It is easy to see that using $ND\text{-}TI$, it is not possible to follow the standard traversal rule to expand the tree of derivations. However in order to treat the control operator cut the standard traversal must be followed.

Example 2. Let P be the program:
$q(X){:-}d, p(X)$; $\quad r(X){:-}d, p(X)$;
$p(X){:-}d'$; $\quad p(X){:-}d''$;
$s(X){:-}d''$.
and G_0 be the goal $:-d_0, q(X), r(Y), s(Y), !$, where $d = d_0 = \varepsilon$, $d' = \{X/a\}$ and $d'' = \{X/b\}$. By executing with $ND\text{-}TI$, $G_1 = :-\{X_1/X\}p(X_1), \Box_1, r(Y), s(Y), !$, $G_2 = :-\{X_2/a\}, \Box_2, \Box_1, r(Y), s(Y), !$, $G_3 = :-\{X_1/a\}, \Box_1, r(Y), s(Y), !$, $G_4 = :-\{X/a\}, r(Y), s(Y), !$ and $G_5 = :-\{Y_1/Y\}, p(Y_1), \Box_3, s(Y), !$ are computed. G_5 is a look-up node to the solutions of $lf(G_1)$ in the table. The solution $\{X_1/a\}$ of $lf(G_1)$ is used to expand $lf(G_5)$ and $G_6 = :-\{Y_1/a\}, \Box_3, s(Y), !$ and $G_7 = :-\{Y/a\}, s(Y), !$ are computed. Then a failure is reached and the computation backtracks to G_5. Since there are no more solutions of $lf(G_1)$ in the table, the

computation cannot continue the expansion of G_5 and thus it backtracks to expand another node. However, branches of the tree that are in fact not expanded by the effect of the cut obtained using the fourth clause of P to expand G_5 are in this way computed.

3 Tabled Computations for Prolog Programs with Cut

In this section we describe the tabled interpreter *St-TI* obtained by adding tabulation to a standard Prolog interpreter. From now on, clauses can contain cuts in their bodies. The presence of cuts complicates the "use" of the table.
When a new generalized goal G is produced and there exists, in the table, an entry with key $lf(G')$ equivalent to $lf(G)$, then three different kinds of relations between G and G' can occur: (1) $lf(G')$ has been completely solved, i.e., it is to the left of $lf(G)$ in the tree; (2) G and G' are in the same derivation and $lf(G)$ is not part of the proof of $lf(G')$, i.e., G does not contain a call-exit marker corresponding to $lf(G')$; (3) $lf(G)$ is part of the proof of $lf(G')$, i.e., G contains a call- exit marker corresponding to $lf(G')$ (this is the case of a recursive call).
In order to treat cuts correctly, the tabulation mechanism of *St-TI* is defined in such a way that, if there is no recursion, a solution list in the table is used to expand a node in the tree only when it is sure that this list has not been (or will not be) shortened because of a cut. In the case of a recursive call, as in point (3), the solutions of $lf(G')$ are used to expand G, independently of whether G' contains cuts or not. This is based on the following argument: the part of the search space of $lf(G')$ explored so far led to the recursive call $lf(G)$. Hence, $lf(G)$ either needs less search than $lf(G')$ (if a cut prunes part of it), or it needs the same search in which case the computation falls into a loop. According to these ideas, the three cases distinguished above are dealt with in the following ways:

(1) If G' does not contain cuts, then the solutions of $lf(G')$ that have been collected in the table are used to expand G. Thus, G becomes a look-up node. Otherwise, if G' contains cuts, then the computation of $lf(G)$ is executed independently of that of $lf(G')$.
(2) In this case, according to the standard traversal rule, $lf(G)$ will be completely solved before $lf(G')$. Therefore, if G does not contain cuts, then the solutions of $lf(G)$ can be used to expand G'. G becomes a solution node and G' is turned into a look-up node (notice that it was a solution node). Moreover, the key $lf(G')$ in T_i is replaced by $lf(G)$, the corresponding solution list is accordingly renamed and a pointer to the end of that list is added to G'. Otherwise, if G contains cuts, then the computation of $lf(G)$ is executed independently of that of $lf(G')$.
(3) G becomes a look-up node independently of whether G or G' contain cuts or not. A pointer to the solution list in the table associated to the key $lf(G')$ is added to G. In this situation, when there are no more solutions of $lf(G')$ to consider for expanding G, a loop is discovered and the whole computation is stopped.

The following examples illustrate the points described above.

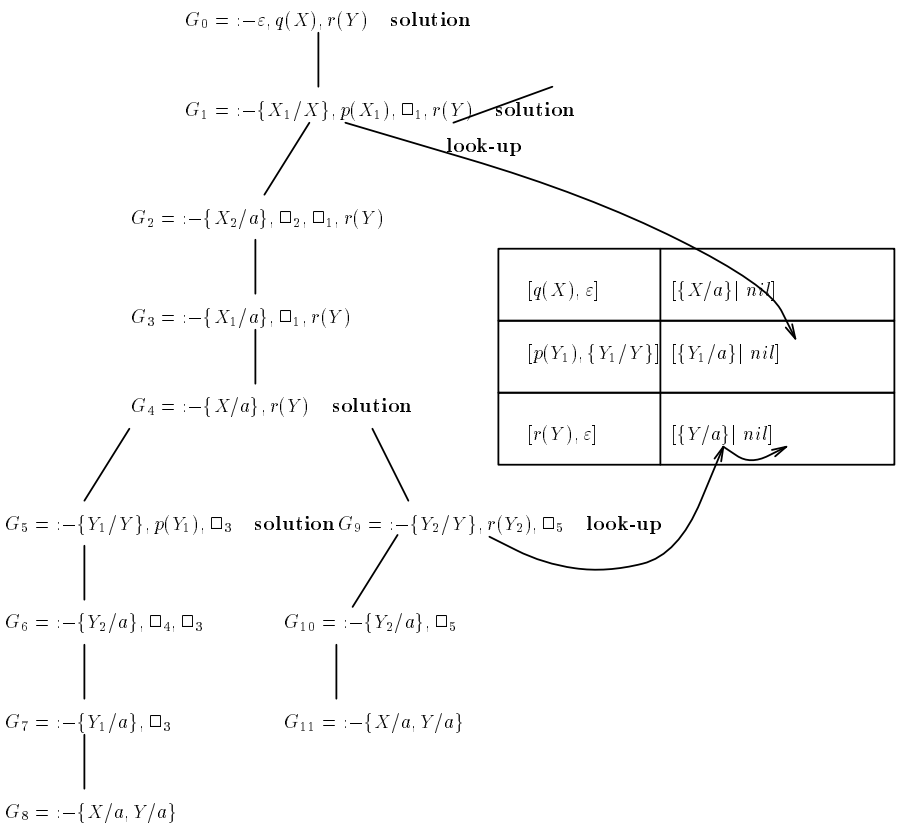


Fig. 2.

Example 3. Let P be the program:
$q(X){:-}\varepsilon, p(X)$; $\quad r(X){:-}\varepsilon, p(X)$;
$r(X){:-}\varepsilon, r(X)$; $\quad p(X){:-}\{X/a\}$.
and G_0 be the goal $:-\varepsilon, q(X), r(Y)$. Starting from G_0, the tree-table pair represented in Fig. 2 is reached.
Observe that, when G_5 is considered, the case (2) described above occurs. In fact, there exists in the table a key, $lf(G_1) = [p(X_1), \{X_1/X\}]$, equivalent to $lf(G_5) = [p(Y_1), \{Y_1/Y\}]$. G_1 and G_5 are in the same derivation and $lf(G_5)$ is not part of the proof of $lf(G_1)$. Since G_5 does not contain cuts, G_5 becomes a solution node and G_1 is turned into a look-up node. The entry in the table with key $lf(G_1)$ is modified as described in point (2) above.
When G_9 is generated, the case (3) is applied. G_4 and G_9 are in the same derivation and $lf(G_9)$ is part of the proof of $lf(G_4)$ (because G_9 contains $\Box_5$ that corresponds to $lf(G_4)$). In this case G_9 becomes a look-up node and a pointer to the solution list associated to the key $lf(G_4)$ in the table is added to G_9. The solution $\{Y/a\}$ is used to expand G_9 and when the computation backtracks to G_9 a loop is discovered. □

Given a tree-table pair (t, T), the table T could contain several entries with equivalent keys. This is because, when a new goal G is produced and there exists in the table an entry with key $lf(G')$ equivalent to $lf(G)$ such that G' contains cuts, then a new entry with key $lf(G)$ could be added to the table. However, this happens only when there is no recursion between G and G' (see point (3) above). This guarantees that, if a finite computation system $\mathcal{D}$ is taadopted, then only a finite number of different entries will be added to the table during the computation. When a new goal G is considered, there may exist in the table several entries with key equivalent to $lf(G)$. However, it is possible to show that at most one of them is usable by G, according to the above description of *St-TI*.

4 Tabled Computations for Prolog Programs with Cut and Their Static Analysis

Using *St-TI* as an interpreter for the Abstract Interpretation of Prolog programs, one may obtain incomplete analysis because of the way *St-TI* handles cuts and loops.

Example 4. In this example P is a program which contains a cut and P' is a corresponding abstract program abstracting its groundness and freeness information. The cut in P is not executed by *St-TI*, whereas that in P' is executed. Because of this, the static analysis results incorrect.
A computation system for representing this information is

$$\mathcal{GF} = (\mathbf{GF}, \Phi_{GF}, \leq_{GF}, \Pi_{GF}, \equiv_{GF})$$

where $\mathbf{GF} = (\wp(\mathbf{V}) \times \wp(\mathbf{V})) \cup \{\bot\}$.
The partial order, $\leq_{GF}$, is defined by: $\forall (G, F), (G_1, F_1), (G_2, F_2) \in \wp(\mathbf{V}) \times \wp(\mathbf{V})$,

$$\bot \leq_{GF} (G, F) \text{ and } (G_1, F_1) \leq_{GF} (G_2, F_2) \text{ iff } G_1 \supseteq G_2 \ \& \ F_1 \supseteq F_2.$$

Computation systems for computing groundness and/or freeness analysis are proposed in cf. [5, 6, 8].
Let P be the concrete Prolog program:
$q(X){:-}\{Y/f(Z)\}, p(Y), !; \quad q(X){:-}\{X/a\}; \quad p(X){:-}\{X/g(a)\}.$
and G_0 be the goal $:-\varepsilon, q(X)$. The computation of P with G_0 by *St-TI* is finite and produces one answer substitution, $\{X/a\}$.

Consider the abstract program P' over $\mathcal{GF}$ corresponding to P:
$q(X){:-}(\emptyset, \{X\}), p(Y), !; \quad q(X){:-}(\{X\}, \emptyset); \quad p(X){:-}(\{X\}, \emptyset).$
with goal $G'_0 = :-(\emptyset, \{X\}), q(X)$. Using a renaming of the first clause in P' to resolve $lf(G'_0)$, one obtains the resolvent $G'_1 = :-(\emptyset, \{X_1\}), p(Y_1), !, \square_1$. Then $lf(G_1)$ is unified with the head of the third clause. The unification succeds and $G'_2 = :-(\{Y_2\}, \emptyset), \square_2, !, \square_1$ is computed. The computation proceeds by executing $\square_2$, the primitive ! and $\square_1$. $G'_3 = :-(\{Y_1\}, \{X_1\}), !, \square_1$, $G'_4 = :-(\{Y_1\}, \{X_1\}), \square_1$ and $G'_5 = :-(\emptyset, \{X\})$ are successively computed.
Because of the execution of the cut, the second clause of P' is not considered and

thus an abstraction of the concrete answer substitution $\{X/a\}$ is not produced. Therefore the analysis concludes incorrectly that variable X of the initial goal is always free at the end of the computation. □

A similar phenomenon can occur when *St-TI* detects a loop during an abstract computation: it is possible that the concrete computation does not fall into the same loop (for instance, because of a previous failure).
In order to treat cuts and check loops correctly when executing an abstract program, information about the sure success of the goals in the concrete computations must be available. We call this knowledge *control information* from now on. Control information can be represented by a complete lattice, CI, consisting in two elements, $\{s, u\}$, where s is for *sure* and u for *unsure* with the ordering $s \leq u$. Adding control information to a domain D (that is another complete lattice) consists in performing the product $D \times CI$, cf. [4]. This amouts to having two copies of each value $d \in D$: one (d, s) and one (d, u). The obtained extended domain will be denoted by D_{ci}. $\mathcal{D}_{ci}$ is the corresponding augmented computation system. The original abstract unification Φ on D is transformed to obtain a function Φ_{ci} on D_{ci} that produces also control information. Φ_{ci}, must be correct, i.e., if it produces s, as control information, then all the corresponding concrete computations are successful. This is always possible (in the worst case u is always produced). However, the quality of the control information computed depends on D.
One may wonder whether it is realistic to assume that control information can be inferred during a static analysis. In [7] it is shown that, with the abstract domain EXP it is possible to infer control information. In particular, the treatment of some Prolog built-ins, can help considerably the inference process.
The computation of *St-TI* on $\mathcal{D}_{ci}$ is illustrated by the following example.

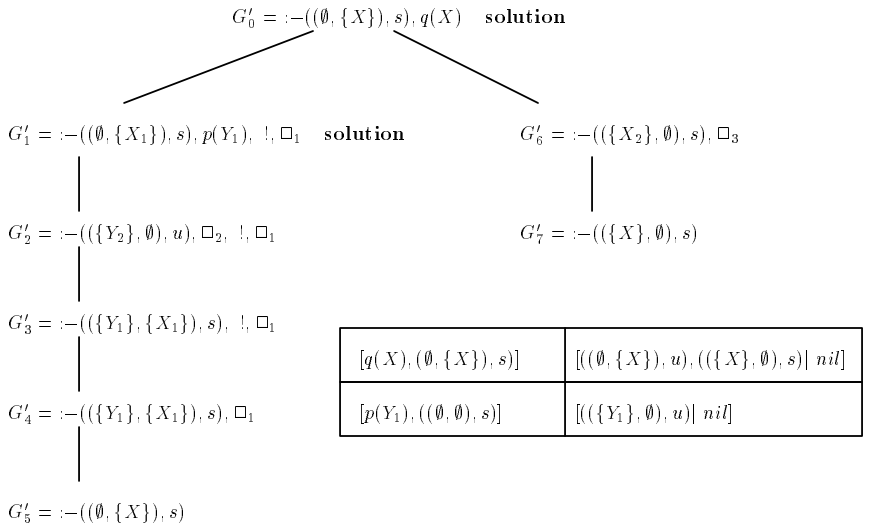


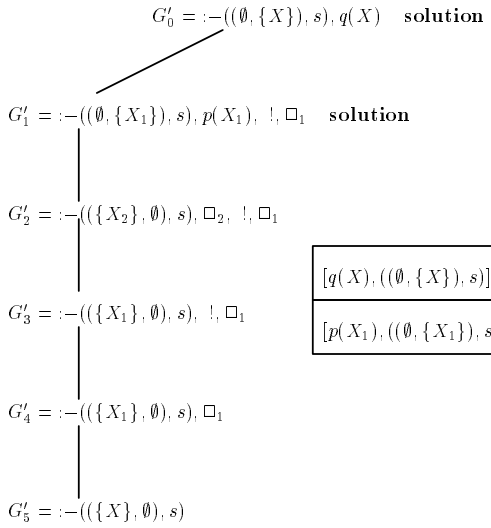
Fig. 3.

Example 5. Let P' and G'_0 be as in the previuos example except that each $(G, F) \in \mathbf{GF}$ is replaced with $((G, F), s) \in \mathbf{GF}_{ci}$ and let $\mathcal{GF}_{ci}$ be the abstract

computation system obtained from $\mathcal{GF}$ by adding to it control information. Executing P' with G'_0 by using the control information to control the execution of the cut, the tree-table pair in Fig. 3 is reached.
G'_1 is $:-((\emptyset,\{X_1\}),s),p(Y_1),!,\Box_1$ and it contains sure information since X and X_1 are both free and thus it is sure that the unification succeeds. However, G'_2 contains unsure information because in G'_1 no information about Y_1 is represented. Then $\Box_2$ is computed and G'_3 is reached. The leftmost element of G'_3 is a cut. In this case we consider all the path from the parent node of this cut (G'_0) to G'_3, looking at the corresponding control information. If the control information in all the goals in this path is sure then we are sure that the cut is executed in all the corresponding concrete computations and thus it can be executed; otherwise the cut is ignored. In this case, the second alternative occurs and thus the cut operator is not executed. An abstraction of the concrete answer substitution $\{X/a\}$ is computed. □

In the following example a cut is reached at the abstract level with sure information. It is executed producing a more efficent and precise analysis.

Example 6. Consider the program obtained by replacing the first clause of P' in the previous example with $q(X):-(\emptyset,\{X\}),p(X),!$. The tree-table pair represented in Fig. 4 is obtained. Observe that, considering the path from G'_0 to G'_3 a sure information is reached at each step. We are sure that the cut is executed in all the corresponding concrete computations and thus we can safely execute it also at the abstract level.



$[q(X),((\emptyset,\{X\}),s)]$	$[((\{X\},\emptyset),s) \mid nil]$
$[p(X_1),((\emptyset,\{X_1\}),s)]$	$[((\{X_1\},\emptyset),s) \mid nil]$

Fig. 4.

□

Acknowledgement

We thank Luca Righi for his help in designing the figures. We thank Giuseppe Nardiello for a careful reading of the paper.

References

1. K. Apt: Introduction to Logic Programming. Handbook of Theoretical Computer Science, J.van Leeuwen, editor, North Holland, 1990.
2. R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi: Modelling Prolog Control. In: Proc. Nineteenth Annual ACM Symposium on Principles of Programming Languages, pp 95–104. ACM Press, 1992.
3. P. Codognet and G. Filé: Computations, abstractions and constraints in logic programs. In: Proc. Fourth International Conference on Programming Languages (ICCL'92), Oakland, U.S.A., April 1992.
4. P. Cousot and R. Cousot: Abstract Interpretation and Application to Logic Programs. Journal of Logic Programming, 13(2&3):103–179, July, 1992.
5. M. Bruynooghe, M. Codish, D. Dams and G. Filé: Freeness analysis for logic programs, 1992. To appear in ICLP'93.
6. A. Cortesi and G. Filé: Abstract Interpretation of Logic Programs: an abstract domain for groundness, equivalence, sharing and freeness analysis. In: N.D. Jones and P. Hudak (eds): ACM Symposium on partial evaluation and semantics based program manipulation. SIGPLAN NOTICES, 26(9), pp. 52–61. 1991.
7. A. Cortesi, G. Filé, and S. Rossi: Abstract Interpretation of Prolog: the treatment of the buit-ins. In: Costantini (ed): Proc. GULP'92, 1992.
8. A. Cortesi, G. Filé, and W. Winsborough: Prop revisited: Propositional Formulas as Abstract Domain for Groundness Analysis. In: G. Kahn (ed): Proceedings of the IEEE sixth annual symposium on Logic In Computer Science (LICS'91), pp. 322–327, Amsterdam, 1991. IEEE Press.
9. S.K. Debray and P. Mishra: Denotational and Operational Semantics for Prolog. In: M. Wirsing (ed): Formal Description of Programming Concepts III, pp. 245–269. North-Holland, Amsterdam, 1987. 1987.
10. T. Kanamori and T. Kawamura: Abstract Interpretation based on OLDT-resolution. ICOT Research Report, Tokyo, July, 1990.
11. B. Le Charlier and S. Rossi: An Accurate Abstract Interpretation Framework for Prolog with cut. Submitted to *ILPS'93*.
12. J.W. Lloyd: Foundations of Logic Programming. Springer, 1987.