Abstract Interpretation of Prolog Programs with Cut and Built-ins*

Agostino Cortesi Gilberto Filé Sabina Rossi
Dept. of Mathematics - University of Padova
Via Belzoni 7, I-35131 Padova ITALY
{cortesi, file, srossi }@pdmat1.unipd.it

Abstract

We are interested in the Abstract Interpretation of real Prolog programs and in particular in handling the control primitive cut at the abstract level.
A cut can be executed during the abstract computation only if it is sure that the same cut is executed in all the corresponding concrete computations. Therefore, in order to execute cuts correctly during the static analysis, information about the sure success or failure of the goals in the concrete computations must be available. We call such information *control information*. Control information can be inferred during the static analysis. In particular, we show that, with the abstract domain EXP, defined in [CFil 91], one can easily infer control information when treating the Prolog built-ins corresponding to tests on the term instantiations. Assuming that control information is inferred during the static analysis, we define a tabled interpreter for the Abstract Interpretation of Prolog programs with cuts and built-ins. It executes a cut whenever the control information guarantees that the same cut is also reached during the concrete computations. This generic fixpoint algorithm always terminates on finite abstract domains.

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. Originally, flow-chart languages were considered [CCou 77]. Starting in the early 80's with the work of Mellish [Mel 81] many researchers studied its application to logic programming [JoSø 87, Bru 88, Deb 89, HeRo 89, MaSø 89, CFWi 91, GiHe 91, CFWi 92, CCou 92] and more recently to concurrent constraint logic languages [MaSø 90, Codo 90, CFM 90].

*Work supported by P.F. Sistemi Informatici e Calcolo Parallelo of CNR, grant 91.00026.69

Every Abstract Interpretation technique works roughly as follows: first one defines a simple (abstract) domain that represents the desired information and secondly one mimics the execution of the programs on this domain.

Our aim is to define a generic fixpoint algorithm for the Abstract Interpretation of real Prolog programs. In particular we are interested in handling the control primitive cut at the abstract level. In order to treat cuts during the abstract computation, information about the sure success of the goals in the concrete computations must be available. In fact, if a cut, that is not executed during the concrete computation (say, because of an earlier failure), is executed during the abstract one, then there may be concrete LD-derivations that are not simulated in the abstract computation, leading to an incomplete static analysis. One simple solution to this problem is to ignore the cuts during the abstract computation. However, it is interesting to be able to execute cuts also at the abstract level in order to eliminate from the abstract computation some useless LD-derivations and to obtain a more efficient and more precise static analysis. Obviously, it is safe to execute a cut during the abstract computation when it is sure that the same cut is executed in all the corresponding concrete computations. Therefore , information about the sure success or failure of the goals in the concrete computations must be inferred at the abstract level. Let us call this knowledge *control information* from now on.

One may wonder whether it is realistic to assume that control information can be inferred during a static analysis. We show that, with the abstract domain *EXP* [CFil 91] representing *groundness* and *covering* [Mel 81, Deb 89, MaSø 89], *sharing* [Deb 89, HeRo 89], *freeness* [Deb 89] and *compoundness* [Mel 81] substitution properties (but not real sophisticated type information), the inference process can be easy. More precisely, treating on EXP the Prolog built-ins that are tests on the term instantiations, control information about "sure success", "sure failure" and "unsure success" of the corresponding concrete tests can be produced. Assuming that control information is inferred during the abstract computation, we define a generic fixpoint algorithm for the Abstract Interpretation of Prolog programs with cuts and built-ins. This is done following the approach of [CoFi 91] in which both the concrete computation of a Prolog (and constraint) program P and its static analysis are obtained uniformly by executing them on different domains with the same general tabled interpreter I. In [CoFi 91], I is obtained by "adding a tabulation mechanism" to an interpreter that nondeterministically searches the SLD trees. Here we start from a standard Prolog interpreter *St-I* and add a tabulation mechanism to it. *St-I*$'$ is the obtained interpreter. *St-I*$'$, in order to execute cuts safely, computes also control information that is consulted every time a cut is encountered in order to decide whether to execute it or to ignore it. *St-I*$'$ is a generic fixpoint algorithm for the static analysis of Prolog programs.

This paper is organized as follows. *Section 1* contains some preliminary definitions. In *Section 2* we show how control information can be inferred by treating some Prolog built-ins on EXP. *Section 3* contains the description of the tabled interpreter *St-I*$'$.

1 Preliminaries

The reader is assumed to be familiar with the basic concepts of Logic Programming, see for instance [Llo 87], [Apt 90]. If A is a relation, $A^\flat$ is the reflexive, symmetric and transitive closure of A. Let $\mathbf{V}$ denote an arbitrarily large finite set of variables; Σ denotes a ranked alphabet of function and predicate symbols. Let $T(\Sigma, \mathbf{V})$ be the set of all terms constructed over Σ and $\mathbf{V}$. If t is a term, or an atom, or a clause, $Var(t)$ denotes the set of all the variables occurring in t. A substitution σ is a function $\mathbf{V} \to T(\Sigma, \mathbf{V})$. *Subst* denotes the set of all idempotent substitutions.
A program over a computation domain D is a finite ordered set of definite clauses of the form:

$$p(\bar{X}){:-}d, A_1, \ldots A_n$$

where $d \in D$, $p(\bar{X})$ is an atom with $\bar{X}$ that is a variable vector, each A_i is either an atom $q(\bar{Y})$ with $\bar{Y}$ that is a variable vector or a built-in or the primitive cut. Analogously a goal over a computation domain D is defined.
Observe that any Prolog program can be put into this form where d will be a substitution.

The abstract domain EXP, defined in [CFil 91], synthesizes the groundness, covering, sharing, boundness and freeness substitution properties. Its elements are 5-tuples of the form $\Delta = (GR_\Delta, C_\Delta, SH_\Delta, B_\Delta, F_\Delta)$ such that $GR_\Delta, B_\Delta, F_\Delta \subseteq \mathbf{V}$, $C_\Delta \subseteq (\wp\mathbf{V} \times \wp\mathbf{V})$, $SH_\Delta \subseteq (\mathbf{V} \times \mathbf{V})$.
The abstraction on EXP of a substitution $\sigma \in Subst$ is the element Δ_σ defined componentwise by:

$$\begin{aligned}
GR_\sigma &= \{x \in \mathbf{V} : Var(\sigma x) = \emptyset\},\\
C_\sigma &= \{(S, S') \in \wp(\mathbf{V}) \times \wp(\mathbf{V}) : Var(\sigma S) \supseteq Var(\sigma S')\},\\
SH_\sigma &= \{(x, y) \in \mathbf{V} \times \mathbf{V} : Var(\sigma x) \cap Var(\sigma y) \neq \emptyset\}^\flat,\\
B_\sigma &= \{x \in \mathbf{V} : \sigma x = f(t_1, \ldots t_m) \text{ for a function } f \text{ of arity } m > 0\},\\
F_\sigma &= \{x \in \mathbf{V} : \sigma x \; is \; a \; variable\};
\end{aligned}$$

and the abstraction of an arbitrary set $\Sigma \subseteq Subst$, $\Sigma \neq \emptyset$, is the element Δ_Σ defined componentwise by:

$$\begin{aligned}
GR_\Sigma &= \cap\{GR_\sigma : \sigma \in \Sigma\},\\
C_\Sigma &= \cap\{C_\sigma : \sigma \in \Sigma\},\\
B_\Sigma &= \cap\{B_\sigma : \sigma \in \Sigma\},\\
SH_\Sigma &= (\cup\{SH_\sigma : \sigma \in \Sigma\})^\flat,\\
F_\Sigma &= \{x \in \mathbf{V} : x \in F_\sigma \forall\sigma \in \Sigma \; and \; if \; (y, x) \in SH_\Sigma \; then \; (\{y\}, \{x\}) \in C_\Sigma\}.
\end{aligned}$$

$\Delta = (GR_\Delta, C_\Delta, SH_\Delta, B_\Delta, F_\Delta)$ abstracts every $\Sigma \subseteq Subst$, $\Sigma \neq \emptyset$ such that $GR_\Sigma \supseteq GR_\Delta, C_\Sigma \supseteq C_\Delta, SH_\Sigma \subseteq SH_\Delta, B_\Sigma \supseteq B_\Delta$ and $F_\Sigma \supseteq F_\Delta$.
The abstract unification function Φ and the function for treating the built-ins on EXP, Θ, are defined as follows. Let γ be the concretization function, σ, σ_1 and $\sigma_2 \in Subst$, A, B be two atoms.
Φ is from $(EXP^2 \times Subst)$ to EXP such that, $\forall\sigma_1 \in \gamma(\Delta_1)$, $\forall\sigma_2 \in \gamma(\Delta_2)$ and $\forall$ A, B with $\delta = mgu\{A = B\}$, then $\Phi(\Delta_1, \Delta_2, \delta) = \Delta'$ abstracts the result of the concrete

unification of the atom A under σ_1 with the atom B under σ_2 (see [CFil 91]).
Θ is from $EXP \times (Built - ins)$ to EXP such that $\forall \sigma \in \gamma(\Delta)$, $\Theta(\Delta, \mathtt{b}) = \Delta'$ abstracts the result of the concrete computation of the built-in `b` under the activation substitution σ (see [CFR 92]).

2 Control Information Inferred by Treating Prolog's Built-ins on EXP

When considering Abstract Interpretation over a domain, like EXP, representing information on the instantiation of the program variables, the sure success or failure of the unification between two atoms can be inferred only in trivial cases. However, it is possible to obtain more interesting control information when one considers some *built-ins* of "real" Prolog [ClMe 84, LPA 88]. The reason of this is that some built-ins are tests on the instantiation of the variables. Hence in these cases, the abstract states of EXP allows us to answer the tests. We illustrate this point in the following example.

Example 2.1 *Consider the behaviour of the goal* `:-var(X)` *under an activation substitution σ. By the declarative semantics of the built-in* `var(X)`*, such a goal fails if* σ`(X)` *is not a free variable, otherwise it succeeds and its answer substitution is exactly σ. No unification is made, but only a test about the freeness of* `X` *with respect to σ.*
Let us mimic this situation at the abstract level. Let $\Delta \in EXP$ be an activation state. Three cases can apply.

(1) *If* `X`$\in (GR_\Delta \cup B_\Delta)$ *then we are certain that* `X` *is not free with respect to any substitution $\sigma \in \gamma(\Delta)$. According to the concrete case, also the abstract computation will signal sure failure.*

(2) *If* `X`$\in F_\Delta$ *then the built-in surely succeeds, because in this case* `X` *is free with respect to all substitutions in $\gamma(\Delta)$.*

(3) *Otherwise, $\gamma(\Delta)$ may contain both substitutions σ such that* `X` *is free with respect to σ and substitutions σ' such that* `X` *is not free with respect to σ'. In both cases Δ would be modified into Δ' stating that the variable* `X` *is free. Thus, whenever it is possible, the resulting abstract state Δ' will be obtained from Δ by modifying only its F-component, that will become $F_\Delta \cup \{\mathtt{X}\}$.*

The example above suggests that the result of an abstract test $\Theta(\Delta \mathtt{b})$, where $\Delta \in EXP$ and `b` is the built-in, can be one of the following three: (i) $(\perp, s)$ meaning "sure failure"; (ii) (Δ, s) meaning "sure success"; (iii) (Δ', u) meaning "unsure success" with outcome the abstract state Δ'. It satisfies the following correctness conditions: in case (i) $\forall \sigma \in \gamma(\Delta)$ the goal $: -\mathtt{b}$, under the activation substitution σ, fails; in case (ii) $\forall \sigma \in \gamma(\Delta)$, the goal $: -\mathtt{b}$, under the activation substitution σ, succeeds and the corresponding answer substitution σ' belongs to $\gamma(\Delta)$; in case (iii) $\forall \sigma \in \gamma(\Delta)$, if the goal $: -\mathtt{b}$, under the activation substitution σ, does not fail and if σ' is the corresponding answer substitution then $\sigma' \in \gamma(\Delta')$.

Thus one can assume that the result of $\Theta(\Delta, \mathtt{b})$ is always a pair (Δ, i), instead of the single value Δ, where the second component i can be s for "sure" or u for "unsure". The information i is called *control information*.
In this way, most of Prolog's built-ins can be treated in EXP obtaining control information from them. For details see [CFR 92].

3 The Tabled Interpreter St-I'

Using control information, we define a tabled interpreter for the static analysis of Prolog programs, treating cuts and built-ins.

Given a goal G, over a domain D of the form $:-d, A_1, \ldots, A_n$ with $A_1 = p(\bar{X})$, then the *left-most call pettern* of G is

$$lf(G) = [p(\bar{X}), \Pi(d, \bar{X})]$$

where $\Pi(d, \bar{X})$ is the projection of d on the variables of $\bar{X}$.
Observe that the left-most call-pattern of a goal is defined only if its left most symbol is an atom.
A *left-most call pettern* $[p(\bar{X}), d]$ with $\bar{X} = (X_1, \ldots, X_n)$ is called *equivalent* to the *left-most call pattern* $[p(\bar{Y}), d']$ with $\bar{Y} = (Y_1, \ldots, Y_n)$, noted $[p(\bar{X}), d] \equiv [p(\bar{Y}), d']$, when $d\{X_1/Y_1, \ldots, X_n/Y_n\} = d'$.

The idea of the tabled computation is as follows: collect in a table all the left-most call patterns of the goals found so far in the computation, and whenever a new goal G is produced, check whether the table already contains a left-most call pattern $lf(G')$ equivalent to $lf(G)$. In this case use for expanding G the solutions of $lf(G')$ that have been collected in the table. G' is called *solution node* and G *look-up node*. In a case this tabulation mechanism is not according to the depth-first left-to-right computation rule. It is when G and G' are goals of the same derivation, but $lf(G)$ is not part of the proof of $lf(G')$. The situation is the following:

where s represents a solution for $lf(G')$. In this case, by means of the tabulation mechanism just described, the solutions of $lf(G')$ are used for expanding G and thus they are all computed before of the corresponding solutions of $lf(G)$. This is not according to the depth-first computation rule, by means of that, in this case G would be completely solved before of G'.
In order to perform a tabulation mechanism according to that rule, when this situation occurs, we change the roles of G and G' as follows. G becomes a *solution node* and G' is turned into a *look-up node*, in such a way that G is completely solved before of

G' and, when the computation backtracks to G', the solutions of $lf(G)$ are used to continue the computation of G.
This new tabulation mechanism can be added to the standard Prolog interpreter, *St-I*. It performs loop-check as follows. Consider the situation:

in which $lf(G) \equiv lf(G')$, $lf(G)$ is part of the proof of $lf(G')$, and $s_1, \ldots, s_n$ are the solutions of $lf(G')$ computed so far. In this case the solutions of $lf(G')$ are used for expanding G, and when there are no more solutions in the table to consider, then a loop is detected. This is because the next alternative for $lf(G)$ would be the same that, in the proof of $lf(G')$, has produced it.
What happens when cuts occur in the goals?
It is clear that, for treating cuts correctly, the solutions in the table associated to a $lf(G')$ such that G' contains cuts, cannot be used for expanding another goal G with equivalent $lf(G)$ that is not part of the proof of $lf(G')$. This is true independently of whether G contains cuts or not. In fact in this case, the solution list of $lf(G')$ could have been shortened because of a cut. Therefore, if G' contains cuts, then *St-I'* solves $lf(G)$ independently of $lf(G')$. However, if $lf(G)$ is part of the proof of $lf(G')$, then *St-I'* uses the solutions of $lf(G')$ for expanding G because if G' contains cuts then they have no effect on the computation of $lf(G)$. Thus the loop-check mechanism is performed also when cuts occur in the goals.
At the abstract level, when a cut becomes the left most element of a goal G, then it can be safely executed only if it is sure that the same cut is executed in all the corresponding concrete derivations.

Example 3.1 *Let P be the program over the concrete domain of substitutions:*

(1) $p(X){:-}r(Y), q(Y), !.$

(2) $p(X){:-}\{X/a\}.$

(3) $q(Y){:-}\{Y/a\}.$

(4) $r(Y){:-}\{Y/b\}.$

and G be the goal $:-p(X)$.

At concrete level the cut is not executed because of a previous failure and the answer substitutions $\{x/a\}$ is computed. Let us now mimic the computation of P with G over the abstract domain $GR = \wp(\mathbf{V})$ obtained from EXP by considering only the first component of its elements. GR sinthesizes the groundness properties of substitutions.

The abstract program P' over GR corresponding to P is:

(1) $p(X){:}{-}\emptyset\ r(Y), q(Y), !.$

(2) $p(X){:}{-}\{X\}.$

(3) $q(Y){:}{-}\{Y\}.$

(4) $r(Y){:}{-}\{Y\}.$

and the abstract goal G' corresponding to G is $:-\emptyset\ p(X)$.
It is easy to see that computing on GR, the cut is executed. In fact the unification of $:-\emptyset\ q(Y)$ *with the head of the third clause succeeds and the computation proceeds by executing the cut. This leads to an incomplete analysis.*
In order to treat cuts correctly at the abstract level, control information must be available. Suppose that control information is inferred. In this example, unifying $:-\emptyset\ p(X)$ *with the head of the first clause, one obtains the resolvent* $:-(\emptyset, s), r(Y), q(Y), !$ *where* s *means that the concrete unification surely succeeds. Then considering the head of the first clause one obtains the resolvent* $:-(\{Y\}, s), q(Y), !$. *However it is not sure that the unification of* $:-(\{Y\}, s), q(Y)$ *with the head of the third clause succeeds in the concrete computation. Therefore the resolvent of this unification is* $:-(\{Y\}, u), !$, *meaning that it is not sure that the cut occurs in the concrete computation. In this case St-I' ignores the cut.*

As shown in the example above, *St-I'* uses the control information that can be inferred during the static analysis to perform the following correctness rule: at the abstract level a cut is executed only when it is sure that the same cut is executed in all the corresponding concrete derivations.
St-I' is a generic fixpoint algorithm for the abstract interpretation of Prolog programs with cuts and built-is.

References

[Apt 90] Apt K.: "Introduction to Logic Programming." in *Handbook of Theoretical Computer Science.* J.van Leeuwen ed., North Holland. 1990.

[Bru 88] Bruynooghe M.: "A practical framework for the abstract interpretation of logic programs." *Journal of Logic Programming* 1992.

[CCou 77] Cousot P., Cousot R.: "Abstract Interpretation: a unified framework for static analysis of programs by construction of approximation of fixpoints". In *Proc. 4th ACM POPL.* 1977.

[CCou 92] Cousot P., Cousot R.: "Abstract Interpretation and Application to Logic Programs". Rapport de Recherche LIENS-92-12. To appear in the special issue on Abstract Interpretation of the *Journal of Logic Programming*. 1992.

[CF 89] Corsini M-M., Filè G.: "A complete framework for the abstract interpretation of logic programs: theory and application." Research Report, Dipartimento di Matematica Universita' di Padova. 1989.

[CFil 91] Cortesi A., Filè G.: "Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness, and compoundness analysis." Proc. ACM-PEPM, P. Hudak and N. D. Jones eds., ACM-SIGPLAN Notices 26 (9). 1991.

[CFR 92] Cortesi A., Filè G., Rossi S.: "Abstract interpretation of Prolog: the Treatment of the built-ins." In *Proc. GULP 92* (87-103), S. Costantini (ed.). Milano 1992.

[CFWi 91] Cortesi, A., Filè, G. and Winsborough W., "Prop revisited: Propositional formula as abstract domain for groundness analysis." *Proc. Sixth Annual IEEE Symposium on Logic In Computer Science.* G. Kahn (ed.), pp.322-327, Amsterdam 1991.

[CFWi 92] Cortesi, A., Filè, G. and Winsborough W., "Comparison of Abstract Interpretations." *Proc. 19th International Colloquium on Automata, Languages and Programming* W.Kuich (ed.), *LNCS 623* (523-535) Springer-Verlag 1992.

[Che 91] Cheong P-H.: *Type inference by abstract interpretation.* Ph.D. dissertation, LIENS, Paris,1991, forthcoming.

[ClMe 84] Clocksin W.F., Mellish C.S.: *Programming in Prolog.* Springer-Verlag, 2nd ed., Berlin, 1984.

[CoFi 91] Codognet P., Filè G.: "Computations, Abstractions and Constraints in Logic Programming." R.I. Università di Padova. 1991.

[Codo 90] Codognet C., Codognet P., Corsini M.: "Abstract Interpretation for concurrent logic languages." In *Proc. NACLP'90*, Austin, 1990.

[CFM 90] Codish M., Falaschi M., Mariott K.: "Suspension Analysis for Concurrent Logic Programs." Research Report - University of Pisa, 1990.

[Deb 89] Debray S.K.: "Static inference of modes and data dependencies in logic programs". In *TOPLAS* Vol.11 (418-450). 1989.

[GiHe 91] Giannotti F., Hermenegildo M.: *A technique for recursive Invariance Detection and Selective Program Specialization.* In Proc. PLILP'91, *LNCS 528* (323-334) Springer-Verlag 1991.

[HeRo 89] Hermenegildo M., Rossi F.: "On the correctness of independent AND-parallelism in logic programs." In *Proc. NACLP'89.* Cleveland 1989.

[JoSø 87] Jones N., Søndergaard H.: "A semantic based framework for the abstract interpretation of Prolog." In *Abstract Interpretation of Declarative Languages*, ed. S.Abramsky and C.Hankin. Ellis Horwood. 1987

[Llo 87] Lloyd J.W.: *Foundations of Logic Programming*. Springer, 1987.

[LPA 88] Clark K. L., McCabe F. G., Johns N., Spenser C., *LPA MacPROLOG Reference Manual* 1988

[MaSø 89] Marriott K., Søndergaard H.: "Notes for a tutorial on Abstract Interpretation of logic programs." NACLP'89. Cleveland 1989.

[MaSø 90] Marriott K., Søndergaard H.: "Analysis of Constraint Logic Programs". In *Proc. NACLP'90*, Austin, 1990.

[Mel 81] Mellish C.S.: "The automatic generation of mode declarations for Prolog programs." In *Proc. Workshop on Logic Prog. for Intel. Systems*. Los Angeles. 1981.