A Denotational Semantics for Prolog

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

Abstract

In this paper we propose a denotational semantics for Prolog and an approach to the abstract interpretation of Prolog programs; we deal with the control rules of Prolog and the cut operator. Moreover, we get a simple denotation for negation as finite failure. The abstract analysis is proposed both for computed answers analysis and for call patterns analysis. In both cases the abstract semantics is finitely computable. Two examples show the usefulness of our approach for the analysis of Prolog programs.

1 Introduction

A semantics can be useful as a tool for program understanding, as a tool for defining an equivalence relation between programs (in relation to program transformations), or as a tool for program analysis. The latter is our main interest. The first question to be answered is why we look for a new semantics of a subset of Prolog (more precisely, of Prolog without database and set operations). There exist many formalizations for subsets of Prolog, and there exists even a formalization for full Prolog [3]; clearly [3] is an operational rather than a denotational semantics. However, we are not religiously bound to denotational semantics; the main problem with operational semantics is that, being goal-dependent, it is not adequate as a basis for goal-independent analysis, where we are concerned with program properties which hold for any valid query. For instance we want to compute groundness or sharing information at some program points, and we want this information to be correct independently from the query. If we are concerned with goal-independent global analysis, a denotational semantics is definitely better than an operational one. The usual argument against denotational semantics is that it is not very good in dealing with all the operational subtleties. This is not a real issue in global analysis since it deals with abstraction, hence there is often a safe way of disregarding those subtleties without losing correctness and precision. Moreover, one of the aims of this paper is just showing how several operational aspects of Prolog can precisely be modeled in a denotational setting.

So why do not we use one of the denotational semantics already defined for various subsets of Prolog [7, 6, 10]? The answer is that none of them is adequate to abstraction. The only one developed for abstract interpretation is the semantics presented in [11]. The only problem with that semantics is that it follows a goal-dependent approach. For a comparison between our approach and theirs, see conclusions.

Our semantics was explicitly designed for abstraction and we will show it can naturally be used for goal-independent global analysis. Handling Prolog control features allows us to get a more precise analysis. In particular our approach deals with the leftmost selection rule, the depth first search rule and the cut operator. Moreover, it is a simple task to "implement" the Prolog `not/1` operator once we have the cut operator. Hence even negation as finite failure is modeled denotationally. Similarly we can implement some other built-in's like `if_then/2` and `if_then_else/3`.

Our results can be considered as the natural evolution of a series of previous results. The first is [8], where an attempt to model divergence is developed in a fixpoint framework. A more adequate semantics for modeling divergence is presented in [4] as an abstraction of a more concrete semantics given in terms of resultants. A resultant $H : -B$, where B is a non empty set of goals, is abstracted into a "divergent atom" $\widetilde{H}$, representing computation which is still in progress and which can possibly remove all the following atoms observed in the computation. Note that the semantic domain consists of sequences rather than sets of atoms, in order to model the relation between atoms implied by the "backtracking" semantics of Prolog. A further development of the semantics in [4] can be found in [12], where the problem of the abstract semantics (useful for program analysis) is tackled, and the new feature of

"downward approximation" of constraints is introduced; A further development of this line of research is [13], where the semantics is extended to deal with the cut operator and an attempt to solve the problem of downward approximations is made; The used semantic domain is very complex, since any constraint is associated to a history and to a cutpoint. In the present work we will show how this information can safely be discarded. Moreover, we will give a much simpler and effective solution to the problem of downward approximation of constraints. Another problem with [13] was that an abstract analysis built on that semantics was not necessarily finite even when defined on finite abstract domain. This important problem is solved in the present work (see section 9).

Another approach strongly related to ours is the one described in [1], where a semantics for logic programs with Prolog control was obtained by "compiling" a Prolog program into an ask/tell language, so that the semantics of the Prolog program can be viewed as the semantics of a concurrent logic program. The main problem with [1] is that the transformation uses constraints like "the execution of this goal terminates", whose abstraction is not trivial. The problem related to the abstraction of these goals, in a finite abstract analysis framework, is the same as the problem of finding an (upward or downward) approximation of SLD trees with control faced in [2], or even as the problem of the approximation of the consistency of constraints faced in [13]. Up to now there was no sensible proposal for control approximation. Hence all these approaches are still theoretical rather than practical. In this paper we follow the approach of [1]. However we use a new kind of constraints, in particular we use "observability" constraints which check whether they are consistent with the constraint store. Moreover, we suggest how to handle these constraints in the abstract case, and we propose a general approach for approximating them, by keeping only the information which is relevant from the abstract point of view, rather than too concrete information like termination.

In order to better understand the ideas underlying the following sections, consider this Prolog program:

```
d(X) :- d(X).
q(X,Y) :- X = 4, !.
q(X,Y) :- d(X), X = Y.
p(X,Y) :- q(X,Y), r(X).
r(X) :- ...
```

If we compute the s-semantics of this program by ignoring the control information, we get the following denotation for `q`: $\{X = 4, X = Y\}$. This is a complete information but an incorrect one, since the answer $X = Y$ can never be found by a Prolog interpreter due to the divergence of `d`. Moreover we loose precision in deriving abstract information. For instance we cannot tell that `X` is always ground in the call `r(X)`, because the only branch of execution for `q` is the first one, which grounds `X`. Finally, we cannot guess that `X` will never be free in the call of `d` in `q`, because otherwise the cut would have excluded the second clause for `q`.

One could wonder whether the above example is an ad-hoc program, developed just for showing the usefulness of control information for improving program analysis. Therefore let us consider a more "realistic" program:

```
select_vars_in_term(X,[X]) :- var(X), !.
select_vars_in_term(A,[]) :- atom(A), !.
select_vars_in_term(F,L) :- F =..[_ Name|Args],
        select_vars_in_list(Args,L).

select_vars_in_list([],[]).
select_vars_in_list([H|T],[H1|T1]) :- select_vars_in_term(H,H1),
        select_vars_in_list(T,T1).
```

Assume we know from some global analysis that the procedure `select_vars_in_term` is always called with its second argument free. If we take into account the control information, it is easy to guess that in the third clause `F` can never be a free variable nor an atom. This would allow us to provide an optimized code which checks only whether `F` is a number.

Roughly speaking, dealing with control allows us to collect not only the information related to the successful execution of a branch of computation, but even that related to the simple observability of it, which should not be hidden by divergence or cut; we will not try to guess when a branch of a computation is observable or not (this would lead to control

approximation) but merely what abstract information can we derive from the observability of a branch of computation. We will try to solve the problem: "if I arrived here, what do I know?" and not the problem "when do I arrive here?". Hence we will add to any constraint, in the following called "kernel" constraint, an observability condition, in the form of an "observability" constraint, which is to be satisfied in order to make the kernel constraint observable.

2 Preliminaries

We assume the reader familiar with basic algebraic structures. A sequence is an ordered collection of elements with repetitions. :: denotes sequence concatenation.

Abstract interpretation [5] is a technique which allows us to statically ("compile time") determine some dynamic ("run time") properties of a program. The idea is that of executing the program on an "abstract" domain, where every element represents a set of elements of the "concrete" domain. This technique is widely used in computer science both for reasoning about the relations between different semantics and for program analysis.

Note that in the program analysis case the abstract domain should be chosen with the aim of making the computation effective. The abstraction (and related approximation) is needed because we know from Rice's theorem that most "interesting" properties of programs are not effectively computable.

One way of formalizing abstract interpretation is by means of Galois connections

Definition 1 *A Galois connection between the posets $\langle P, \sqsubseteq \rangle$ and $\langle P^a, \sqsubseteq^a \rangle$ is a pair $\langle \alpha, \gamma \rangle$ of total maps such that for all $p \in P$ and for all $p^a \in P^a$ we have $\alpha(p) \sqsubseteq^a p^a$ if and only if $p \sqsubseteq \gamma(p^a)$. α and γ are the abstraction and the concretization maps of the connection.*

We know from a theorem in [5] that, given two complete lattices $\langle P, \sqsubseteq \rangle$ and $\langle P^a, \sqsubseteq^a \rangle$, if $\langle \alpha, \gamma \rangle$ is a Galois connection between them, $\phi : P \mapsto P$ and $\phi^a : P^a \mapsto P^a$ are two monotonic operators and $\alpha(-) = -^a$, then the local correctness condition implies the global one, i.e. $\alpha \circ \phi \sqsubseteq^a \phi^a \circ \alpha$ implies $\alpha(lfp(\phi)) \sqsubseteq^a lfp(\phi^a)$ and moreover $\alpha \circ \phi = \phi^a \circ \alpha$ implies $\alpha(lfp(\phi)) = lfp(\phi^a)$. The following result will allow us to get simpler proofs.

Proposition 2 *Let $\langle F, \sqsubseteq \rangle$ and $\langle F^a, \sqsubseteq^a \rangle$ be two posets and $T : F \mapsto F$, $T^a : F^a \mapsto F^a$ and $\alpha : F \mapsto F^a$ be three monotonic maps such that $\alpha(-) = -^a$ and $\alpha \circ T \sqsubseteq^a T^a \circ \alpha$. Then we can extend F and F^a to two complete lattices $\widetilde{F}$ and $\widetilde{F^a}$ respectively and α, T and T^a to monotonic maps $\widetilde{\alpha} : \widetilde{F} \mapsto \widetilde{F^a}$, $\widetilde{T} : \widetilde{F} \mapsto \widetilde{F}$ and $\widetilde{T^a} : \widetilde{F^a} \mapsto \widetilde{F^a}$ such that $\langle \widetilde{\alpha}, \alpha^{-1} \rangle$ is a Galois connection between $\widetilde{F}$ and $\widetilde{F^a}$, $\widetilde{\alpha}(-) = -^a$ and the correctness condition $\widetilde{\alpha} \circ \widetilde{T} \sqsubseteq^a \widetilde{T^a} \circ \widetilde{\alpha}$ holds.*

The relevance of the above proposition is that we do not need to be concerned with the infinite elements of the two lattices, neither we have to define the semantic operators and the abstraction map on them. Roughly speaking, F consists of the finite elements of $\widetilde{F}$ and F^a consists of the finite elements of $\widetilde{F^a}$.

In the following we will use an "abstract" syntax for Prolog programs, which simplifies the semantic operators. The translation from Prolog into our syntax is straightforward and can be understood by noting that the Prolog clause `q(X):-p(X),!,s(X).` is translated into `q(X):-cut(p(X)) and s(X).` . Finally, our abstract syntax assumes all predicates to be unary; this constraint simplifies the definition of the semantics without loss of generality. The extension of that definition to the general case is anyway straightforward.

3 The denotational semantics

Our semantics will be a denotational one: we will compositionally define the element of the domain representing the meaning of a goal from the meanings of the components of this goal. This allows us to define the semantics of a program as the fixpoint of a suitable continuous operator on the semantic domain.

Definition 3 *Given a program P, its immediate consequences operator is defined as $T_P(I)(\mathtt{p}) = \exists_{\mathtt{x}}([\![\delta_{\alpha,\mathtt{x}}]\!] \otimes \mathrm{i}(\mathcal{E}[\![B]\!]I))$, where `p(x):-B.` is the definition of `p` in P, I is an interpretation, i.e., a map that tells us what we already know about the semantics of the more general goals, and*

$$\mathcal{E}[\![c]\!]I = [\![c]\!] \qquad \mathcal{E}[\![G_1 \texttt{ and } G_2]\!]I = \mathcal{E}[\![G_1]\!]I \otimes \mathcal{E}[\![G_2]\!]I$$

$$\mathcal{E}[\![G_1 \texttt{ or } G_2]\!]I = \mathcal{E}[\![G_1]\!]I \oplus \mathcal{E}[\![G_2]\!]I \qquad \mathcal{E}[\![\mathtt{p(x)}]\!]I = \exists_\alpha([\![\delta_{\mathtt{x},\alpha}]\!] \otimes I(\mathtt{p}))$$

$$\mathcal{E}[\![\mathtt{cut}(G)]\!]I = !(\mathcal{E}[\![G]\!]I) \qquad \mathcal{E}[\![\texttt{exists x.}G]\!] = \exists_{\mathtt{x}}(\mathcal{E}[\![G]\!]I) \ .$$

The semantics of a program P is defined as: $\mathcal{S}(P) = \bigsqcup_{i\geq 0}(T_P^i(I^0))$.

The semantic operators ($\otimes$, $\oplus$, $\exists$, ! and ¡) will be defined in the following. Let us just note that we use a map $[\![\_]\!]$ to build the denotation of a single constraint, while a compositional definition is used for conjunction of atoms ($\otimes$) and disjunction of clauses ($\oplus$). The denotation of a procedure call is obtained by modifying the denotation in the environment I in such a way that all references to α are replaced by references to x. α is the standard variable used as parameter in the denotations of all procedure declarations, while we want a denotation for the specific procedure call.

The denotation for `cut`(G) is obtained by applying the ! operator to the denotation for G. We will describe this operator later. We can just note some of its properties. First of all, it must allow a solution found in the execution of G to "cut away" all the solutions found in the following. Moreover, if we consider the computation of the denotation for `cut`(G) `or` G', the same solution must cut all the solutions found in the execution of G'. This property of expanding the scoping of the cut must be added by the ! operator to the denotation of G to obtain the denotation of `cut`(G). Note however that the scope of the ! built-in in Prolog does not extend beyond the procedure definition it belongs to. Therefore we must remove the property added to the denotation of G by the ! operator when we compute the denotation of a procedure which contains `cut`(G). This is exactly the role of the ¡ operator, used in the definition of T_P. In a given interpretation, all scopes are closed.

In the following section we will describe our semantic domain, while in section 5 we will give the definition of the semantic operators on this domain.

4 Semantic domain

We now define the semantic domain. The lattice of basic constraints could be thought of as the domain of analysis, for instance equations over rational trees, though more abstract "domains" could be used (and they actually will in section 8). The diagonal elements $\delta_{x,y}$ represent unification of x and y. For instance, in the case of rational trees, they represent the equation $x = y$. They are needed to perform parameter passing. The cylindrification operators $\exists_x$ are used to remove from a constraint all the information related to x, and are a simple way for avoiding all renaming problems.

Observability constraints are a new concept; roughly speaking, an observability constraint o is satisfied in a given constraint store S if and only if it is consistent with it, i.e. if and only if $S \wedge o$ is satisfiable. Note that we do not require o to be entailed by S. From an alternative point of view, we can look at observability constraints as constraints which give us some information on the constraint store. If o is satisfied in S then S is consistent with o. This remark will be useful when we will consider abstract observability constraints.

In the following, we will consider pairs consisting of a "kernel" constraints and its "observability" part. A kernel constraint k is observable in a given constraint store if and only if its observability part o is satisfied. The kernel part represents the contribution of the constraint to the constraint store in the case the constraint is observable.

Definition 4 *A basic constraint is an element of a lattice $\langle \mathcal{B}, \leq, \vee, \wedge, true, false\rangle$, where $\vee$ is the greatest lower bound operator, $\wedge$ is the least upper bound operator, true is the top of the lattice and false is the bottom of the lattice. We assume there exist elements $\delta_{x,y} \in \mathcal{B}$: for instance $\delta_{x,y}$ represents the constraint identifying the variables x and y. Moreover, we assume there is a family of monotonic operators $\exists_x$ on the set of constraints, representing the restriction of a constraint obtained by hiding all the information related to the variable x.*

The set of kernel constraints is defined as $\mathcal{K} = \mathcal{B}$.

The set $\mathcal{O}$ of observability constraints is defined as follows:

- *a basic constraint is an observability constraint;*
- *if a_1 and a_2 are observability constraints, then $a_1 \sqcap a_2$ is an observability constraint;*
- *if a_1 and a_2 are observability constraints, then $a_1 \sqcup a_2$ is an observability constraint;*
- *if a is an observability constraint, then $-a$ is an observability constraint.*

We assume an injection map $\propto obs$ from kernel constraints into observability constraints defined as $k \propto obs = k$.

A conditional convergent constraint (convergent constraint for short) is an element of $\mathcal{C} = \mathcal{O} \times \mathcal{K}$. $o + k$ will denote $\langle o, k\rangle \in \mathcal{C}$, where o is the observability part and k is the kernel

part of the constraint. We will use the usual notation for field selection: $(o+k)_1 = o$ *and* $(o+k)_2 = k$.

We define the following sets of constraints: $\widetilde{\mathcal{C}} = \{\widetilde{o+k} | o+k \in \mathcal{C}\}$ *(divergent constraints),* $\underline{\mathcal{C}} = \{\underline{o+k} | o+k \in \mathcal{C}\}$ *(open convergent constraints) and* $\overline{\underline{\mathcal{C}}} = \{\overline{\underline{o+k}} | o+k \in \mathcal{C}\}$ *(open internal constraints). A constraint is an element of the set* $\mathfrak{C} = \mathcal{C} \cup \widetilde{\mathcal{C}} \cup \underline{\mathcal{C}} \cup \overline{\underline{\mathcal{C}}}$. *The module operator* $|\cdot| : \mathfrak{C} \mapsto \mathcal{C}$ *is defined as* $|o+k| = o+k$, $|\widetilde{o+k}| = o+k$, $|\underline{o+k}| = o+k$ *and* $|\overline{\underline{o+k}}| = o+k$.

Definition 5 *The usual notion of satisfiability is defined on kernel constraints. A kernel constraint* k *is satisfiable in a constraint store* S *and in a structure interpretation* $\mathfrak{I}$*, if and only if there exists an environment* ρ *such that* $\models^{\rho}_{\mathfrak{I}} (S \wedge k)$.

Satisfiability for observability constraints is different in that we allow different environments to be used in different proofs.

- *if* $o \in \mathcal{B}$*, then* o *is satisfiable in* S *and* $\mathfrak{I}$*, if and only if* o *is satisfiable in* S *and* $\mathfrak{I}$ *as a kernel constraint;*
- $o_1 \sqcap o_2$ *is satisfiable in* S *and* $\mathfrak{I}$*, if and only if both* o_1 *and* o_2 *are satisfiable in* S *and* $\mathfrak{I}$*;*
- $o_1 \sqcup o_2$ *is satisfiable in* S *and* $\mathfrak{I}$*, if and only if* o_1 *or* o_2 *is satisfiable in* S *and* $\mathfrak{I}$*;*
- $-o$ *is satisfiable in* S *and* $\mathfrak{I}$*, if and only if* o *is not satisfiable in* S *and* $\mathfrak{I}$.

The lifting of an observability constraint with respect to a kernel constraint is the observability constraint which already "knows" a bit of the constraint store it will be evaluated in. More precisely, $k \bullet o$ is satisfied in S, if and only if o is satisfied in $S \wedge k$.

Definition 6 *The lifting of an observability constraint with respect to a kernel constraint is defined as follows:*

$$k \bullet k' = k \wedge k' \qquad k \bullet (o_1 \sqcap o_2) = (k \bullet o_1) \sqcap (k \bullet o_2)$$
$$k \bullet (o_1 \sqcup o_2) = (k \bullet o_1) \sqcup (k \bullet o_2) \qquad k \bullet -o = -(k \bullet o)$$

Definition 7 *Given a sequence of constraints* s*, we define its divergence condition as:*

$$\delta(s) = \bigsqcup_{\widetilde{o+k} \in s} o \sqcap (k \propto obs)$$

its cut condition as:

$$\kappa(s) = \bigsqcup_{\underline{o+k} \in s} (o \sqcap (k \propto obs)) \sqcup \bigsqcup_{\overline{\underline{o+k}} \in s} (o \sqcap (k \propto obs))$$

its block condition as $\beta(s) = \delta(s) \sqcup \kappa(s)$ *and its convergence condition as:*

$$o(s) = \bigsqcup_{o+k \in s} (o \sqcap (k \propto obs)) \sqcup \bigsqcup_{\underline{o+k} \in s} (o \sqcap (k \propto obs))$$

We will develop a semantics scheme for Prolog programs which assigns a sequence of constraints as denotation for a goal; we will make things going in such a way that the divergence condition related to this sequence checks whether the computation diverges starting from a given constraint store or not; similarly the cut condition checks whether the computation of subsequent constraints is blocked by a cut or not; the convergence condition checks whether the computation reaches a solution or not (it may even diverge, but only after reaching at the least a solution). We write $\mathcal{SS}$ for the set of sequences of constraints.

The conditioning of a sequence of constraints improves the observability part of them requiring an additional condition to be satisfied.

Definition 8 *The conditioning of a sequence with an observability constraint is defined as* $o \cdot \langle\rangle = \langle\rangle$ *and* $o \cdot \langle v_1, \ldots, v_n \rangle = \langle o \cdot v_1, \ldots, o \cdot v_n \rangle$*, where*

$$o \cdot \langle \widetilde{o'+k'} \rangle = \langle \widetilde{o \sqcap o'+k'} \rangle \qquad o \cdot \langle \underline{o'+k'} \rangle = \langle \underline{o \sqcap o'+k'} \rangle$$
$$o \cdot \langle \overline{\underline{o'+k'}} \rangle = \langle \overline{\underline{o \sqcap o'+k'}} \rangle \qquad o \cdot \langle o'+k' \rangle = \langle o \sqcap o'+k' \rangle \,.$$

Given a sequence of constraints, its instantiation with a kernel constraint is a sequence whose constraints are evaluated in a constraint store which already contains the kernel constraint: so the observability part of the constraints is lifted, while the kernel part is expanded with the kernel constraint.

Definition 9 *The instantiation of a sequence with a kernel constraint is defined as* $k \circ \langle\rangle = \langle\rangle$ *and* $k \circ \langle v_1, \ldots, v_n\rangle = \langle k \circ v_1, \ldots, k \circ v_n\rangle$*, where*

$$k \circ \langle \widetilde{o' + k'} \rangle = \langle k \bullet \widetilde{o' + k} \wedge k' \rangle \qquad k \circ \langle \underline{o' + k'} \rangle = \langle \underline{k \bullet o' + k \wedge k'} \rangle$$
$$k \circ \langle \overline{\underline{o' + k'}} \rangle = \langle \overline{\underline{k \bullet o' + k \wedge k'}} \rangle \qquad k \circ \langle o' + k' \rangle = \langle k \bullet o' + k \wedge k' \rangle \ .$$

We are now ready to define the semantic operators of definition 3.

5 Semantic operators and their properties

Given a syntactic constraint c, its denotation is simply c, with no observability condition (i.e. the observability condition is always satisfiable): $[\![c]\!] = true + c$.

Sum of sequences is not simply sequence juxtaposition. Clearly the first sequence must precede the second, because its constraints are observed before the ones of the latter. However we must be sure that if the computation stops in the first sequence, due for instance to divergence or cut, all the constraints of the second sequence are not observable. Therefore we instantiate them with the opposite of the block condition of the first sequence. The rationale is that they are observable only if the computation did not stop in the first sequence.

Definition 10 *Given two sequences of constraints* s_1 *and* s_2*, we define their sum as* $s_1 \oplus s_2 = s_1 :: -\beta(s_1) \cdot s_2$.

The cut of a denotation should allow the first observable and satisfied constraint to cut all the following constraints. Therefore we add the opposite of the convergence condition of the leading portion of a sequence to the observability part of the trailing portion of it. If the constraints of the trailing portion are observable, then no solution was found in the first portion of the sequence (because its convergent constraints were inconsistent or not observable). Note however that, if the sequence obtained by the application of the cut operator is used as the left operand of a $\oplus$ operation, we must allow the cut to extend its scope to the right operand (because the cut operator cuts horizontally all the solutions found in the portion to its left of the clause it appears in and cuts vertically all the solutions found using the following clauses). Hence all the convergent constraints are transformed into open convergent constraints, so that they can contribute to the definition of β (see above definitions of β and $\oplus$).

Definition 11 *The map* $!$ *on sequences is defined as follows*

$$!(\langle o + k\rangle) = \langle \underline{o + k} \rangle \qquad !(\langle \underline{o + k}\rangle) = \langle \underline{o + k} \rangle$$
$$!(\langle \overline{\underline{o + k}}\rangle) = \langle \overline{\underline{o + k}} \rangle \qquad !(\langle \widetilde{o + k}\rangle) = \langle \widetilde{o + k} \rangle \ .$$

Moreover, if $length(s) \geq 2$ *and therefore* $s = s_1 :: s_2$*, where* s_1 *and* s_2 *are non empty sequences, we define* $!(s) = !(s_1) :: -o(s_1) \cdot !(s_2)$.

The ¡ map is used to "freeze" a denotation. The scope of the cut contained in the denotation must be closed, thus preventing it from including any constraints which can be added by the $\oplus$ operator. We do it by transforming all open convergent constraints into convergent constraints, which will not contribute anymore to β (see definition of $\oplus$) and by removing all the open internal constraints, because they are not needed anymore. Their scope must be closed and moreover they do not belong to the set of convergent constraints needed to define the observable set.

Definition 12 *The map* ¡ *on sequences is defined as follows:*

$$¡(\langle o + k\rangle) = \langle o + k \rangle \qquad ¡(\langle \underline{o + k}\rangle) = \langle o + k \rangle$$
$$¡(\langle \overline{\underline{o + k}}\rangle) = \langle\rangle \qquad ¡(\langle \widetilde{o + k}\rangle) = \langle \widetilde{o + k} \rangle \ .$$

Moreover, if $length(s) \geq 2$ *and thus* $s = s_1 :: s_2$*, where* s_1 *and* s_2 *are non empty sequences, we define* $¡(s) = ¡(s_1) :: ¡(s_2)$.

We extend cylindrification on constraints in the obvious way.

Definition 13 *Given a variable* x*, the map* $\exists_x$ *on sequences is defined as follows*

$$\exists_x(\langle o + k\rangle) = \langle \exists_x o + \exists_x k \rangle \qquad \exists_x(\langle \underline{o + k}\rangle) = \langle \underline{\exists_x o + \exists_x k} \rangle$$
$$\exists_x(\langle \overline{\underline{o + k}}\rangle) = \langle \overline{\underline{\exists_x o + \exists_x k}} \rangle \qquad \exists_x(\langle \widetilde{o + k}\rangle) = \langle \exists_x \widetilde{o + \exists_x k} \rangle \ .$$

Moreover, if $length(s) \geq 2$ *and thus* $s = s_1 :: s_2$*, where* s_1 *and* s_2 *are non empty sequences, we define* $\exists\!\!\!\exists_x(s) = \exists\!\!\!\exists_x(s_1) :: \exists\!\!\!\exists_x(s_2)$.

The following definition sounds very complex and requires careful motivation. Assume we have a denotation D_1 for G_1 and a denotation D_2 for G_2; we look for a denotation D of G_1 `and` G_2. Every convergent constraint in D_1 represents a branch of the execution of G_1 leading to a solution. We do not know and do not care whether this solution is satisfiable or not. All we know is that the empty goal has been reached. The leftmost selection rule of Prolog forces us to execute G_2 in the "context" of this solution. That is how we obtain the third case of the following definition. Something similar happens for open convergent constraints of D_1. They represent a branch of the execution of G_1 leading to a solution. Moreover they represent the fact that if they are observable and satisfiable (i.e. if their observability constraint is satisfied and their kernel constraint is satisfiable in a given constraint store), then all the following constraints must be cut, even if these constraints come after a $\oplus$ operation having D_1 as first operand. We cannot discard this second information and we cannot allow constraints of D_2 to contribute to this cut condition. A cut in G_1 is unaffected by the computation in G_2 even if we execute G_1 `and` G_2. The right solution is to maintain the cut condition as an open internal constraint, where internal means that it does not belong to the frontier of the SLD tree, otherwise it would be a computed answer, while it is a partial answer whose satisfiability fires the cut. This explains the fourth case of the following definition. Moreover, since open internal constraints are not solutions but partial answers, they are unaffected by multiplication. A partial answer for G_1 is even a partial answer for G_1 `and` G_2. This explains the second case of the following definition. For the first case, you should look at a divergent constraint as a branch of the execution of G_1 which must be still expanded. We do not know whether this expansion will eventually lead to a solution or not. Since the leftmost selection rule implies that this "in progress" computation must not be affected by G_2, we leave this constraint untouched.

Let us now consider the recursive case. By performing the multiplication of the first portion of D_1 with D_2, some convergent or open convergent constraint will be substituted by the composition with D_2, according to the third and the fourth cases of the following definition. However, it could be the case that some divergent constraint or some open constraint of D_2 replaces a convergent or open convergent constraint, thus moving all the following constraints within their scope. We must be sure that all the following constraints in the product expand their observability part to take this situation into account. This is done through the ξ map.

Definition 14 *The product of two sequences is defined as follows:*

$$\langle \widetilde{o+k} \rangle \otimes s = \langle \widetilde{o+k} \rangle \qquad \langle \overline{\underline{o+k}} \rangle \otimes s = \langle \overline{\underline{o+k}} \rangle$$
$$\langle o+k \rangle \otimes s = o \cdot (k \circ s) \qquad \langle \underline{o+k} \rangle \otimes s = \langle \overline{\underline{o+k}} \rangle :: o \cdot (k \circ s)\ .$$

Moreover, if $length(s') \geq 2$ *then there exist non empty sequences* s_1 *and* s_2*, such that* $s' = s_1 :: s_2$*. In this case we define* $s' \otimes s = s_1 \otimes s :: -\xi(s_1, s) \cdot (s_2 \otimes s)$*, where* $\xi(s_1, s) = \bigsqcup_{o+k \in s_1} o \sqcap (k \bullet \beta(s)) \sqcup \bigsqcup_{\underline{o+k} \in s_1} o \sqcap (k \bullet \beta(s))$.

We defined all the semantic operators used in the general scheme of definition 3. Let us now consider how to actually compute the semantics.

6 Back to denotational semantics

An interpretation is a map I from the set of predicate symbols Π to the set of sequences of constraints. We can define a partial ordering on the set of sequences which corresponds to the computational evolution of the fixpoint computation. This partial ordering can be elementwise extended to interpretations. The details cannot be described here. The bottom interpretation is the interpretation I^0, such that $I^0[\mathtt{p}] = \langle \widetilde{true + true} \rangle$ for every predicate $\mathtt{p}$. Given a program P, T_P is monotonic on interpretations.

Note that our semantics is defined as $\mathcal{S}(P) = \bigsqcup_{i \geq 0} T_P^i(I^0)$. Actually, while divergent or cut constraints are useful for a precise and compositional definition of the observability conditions, we are not interested in them when we turn to the use of the collected abstract information. In this second step, we only need to know the abstract information of every convergent constraint and the success condition of its related observability constraint. Therefore it is sensible to discard all the redundant information (from this point of view), through

an auxiliary function $\mathcal{O}$, defined as $\mathcal{O}(s)[k] = \bigsqcup_{o+k \in s} o \sqcup \bigsqcup_{\overline{o+k} \in s} o$. $\mathcal{O}$ is extended to interpretations in the obvious way. Namely, $(\mathcal{O}(I))[\mathtt{p}] = \mathcal{O}(I[\mathtt{p}])$ for every predicate $\mathtt{p}$. Roughly speaking, $\mathcal{O}(s)[k] = o$ means that, if k is a computed answer constraint in a given constraint store, then o is satisfiable in that constraint store. Note that, if s contains two constraints $o_1 + k$ and $o_2 + k$ and if o_1 entails o_2, then $\mathcal{O}(s)[k] = o_2$, because, when k is computed, we do not know whether it has been computed by the first or by the second constraint. Hence the most general information we know is that when it is computed then it is computed in a constraint store which satisfies o_2. The pointwise partial order is defined on the range of $\mathcal{O}$, which reflects the fact that, as the computation evolves, the observability condition for a given kernel constraint k becomes higher and higher.

The following is a soundness result.

Theorem 15 *A kernel constraint k is a computed answer constraint for a goal G executed in a program P if and only if $\mathcal{O}(\mathcal{E}[\![G]\!](\mathcal{S}(P)))[k]$ is a satisfiable observability constraint.*

7 A call patterns semantics

In the above section we have shown a semantics able to characterize computed answers of a Prolog program. However program analysis is more often concerned with call patterns, more precisely with the set of all possible call patterns for any predicate. This information is useful for an optimizing compiler, since it allows one to compile predicate calls, by specializing them for the specific call patterns which can actually arise at run-time.

In this section we sketch the definitions of the operators for a call pattern semantics for Prolog. We omit some details, since they can easily be reconstructed as an extension of the case of the computed answers semantics.

The main difference between call patterns and computed answers is that call patterns belong to the internal part of the SLD tree, and not only to its frontier. Moreover, a call pattern constraint is associated to a specific predicate. Hence we define the following kinds of constraint: $\widetilde{o+k;\mathtt{p}}$ (divergent constraints), $o+k$ (convergent constraints), $\overline{o+k;\mathtt{p}}$ (internal constraints), $\underline{o+k}$ (cut constraints) and $\underline{\overline{o+k}}$ (cut internal constraints). Note the introduction of a new kind of internal constraint. The definitions of the semantic operators is similar to the case of the computed answers semantics, with a few relevant differences: $[\![c]\!] = true + c$, $s_1 \oplus s_2 = s_1 :: -\beta(s_1) \cdot s_2$. The definition of $\exists_x$ is essentially unchanged while the basic cases of the definition of the ! operator are the following.

$$
\begin{aligned}
!(\langle\rangle) &= \langle\rangle & !(\langle o+k\rangle) &= \langle \underline{o+k}\rangle \\
!(\langle\overline{o+k;\mathtt{p}}\rangle) &= \langle\overline{o+k;\mathtt{p}}\rangle & !(\langle\underline{o+k}\rangle) &= \langle\underline{o+k}\rangle \\
!(\langle\underline{\overline{o+k}}\rangle) &= \langle\underline{\overline{o+k}}\rangle & !(\langle\widetilde{o+k;\mathtt{p}}\rangle) &= \langle\widetilde{o+k;\mathtt{p}}\rangle\ .
\end{aligned}
$$

The only relevant difference is that internal constraints are left untouched, because a constraint is allowed to cut only if it is a solution of the goal, i.e. only if it is a convergent constraint (which is made open).

Even the basic cases of the definition of ¡ are an extension of the old definition.

$$
\begin{aligned}
¡(\langle\rangle) &= \langle\rangle & ¡(\langle o+k\rangle) &= \langle o+k\rangle \\
¡(\langle\overline{o+k;\mathtt{p}}\rangle) &= \langle\overline{o+k;\mathtt{p}}\rangle & ¡(\langle\underline{o+k}\rangle) &= \langle o+k\rangle \\
¡(\langle\underline{\overline{o+k}}\rangle) &= \langle\rangle & ¡(\langle\widetilde{o+k;\mathtt{p}}\rangle) &= \langle\widetilde{o+k;\mathtt{p}}\rangle\ .
\end{aligned}
$$

Finally the basic cases of the definition of $\otimes$ are

$$
\begin{aligned}
\langle\rangle \otimes s &= \langle\rangle & \langle\widetilde{o+k;\mathtt{p}}\rangle \otimes s &= \langle\widetilde{o+k;\mathtt{p}}\rangle \\
\langle\overline{o+k;\mathtt{p}}\rangle \otimes s &= \langle\overline{o+k;\mathtt{p}}\rangle & \langle\underline{\overline{o+k}}\rangle \otimes s &= \langle\underline{\overline{o+k}}\rangle \\
\langle o+k\rangle \otimes s &= o \cdot (k \circ s) & \langle\underline{o+k}\rangle \otimes s &= \langle\underline{\overline{o+k}}\rangle :: o \cdot (k \circ s)\ .
\end{aligned}
$$

A difference is found in the definition of the T_P operator. Roughly speaking, in the denotation of `p(x)` there exists a call pattern which is precisely `p(x)`, and then there are all the call patterns which arise in its execution. This leads to the following definition: $T_P(I)(\mathtt{p}) = \langle\overline{true + true;\mathtt{p}}\rangle :: \exists_{\mathtt{x}}([\![\delta_{\alpha,\mathtt{x}}]\!] \otimes ¡(\mathcal{E}[\![B]\!]I))$. With this simple changes, the semantics

scheme of definition 3 gives rise to a call patterns semantics for Prolog. It is computed as the least fixpoint of the T_P operator, starting from the environment I^0 such that $I^0[\mathtt{p}] = \langle \widetilde{true + true}; \mathtt{p} \rangle$, for every predicate p.

Denoting the extension of the $\mathcal{E}$ map for call patterns analysis as $\mathcal{E}^{cp}$ and the induced semantics as $\mathcal{S}^{cp}$, we get the soundness result

Theorem 16 *A kernel constraint k is a call pattern for the predicate* p *arising from the execution of the goal G in a program P if and only if $\mathcal{O}(\mathcal{E}^{cp}[\![G]\!](\mathcal{S}^{cp}(P)))[\langle k, \mathtt{p} \rangle]$ is a satisfiable observability constraint.*

8 Abstract semantics

Consider a concrete observability or kernel constraint c. If it is satisfiable in the current store, then we know that c is satisfiable in the current store. Rather than being a silly observation, this remark suggests us to look for a suitable approximation of the "success condition" of c. The most concrete success condition is obviously c itself. Similarly, the most concrete "failure condition" for c is $-c$, that is c is not satisfiable in the current store. So every concrete constraint is isomorphic to a success/failure pair $\langle c, -c \rangle$, which we will write: $\left[\begin{smallmatrix} c \\ -c \end{smallmatrix}\right]$. This is the most concrete success/failure approximation of c. Actually, it is a precise approximation of c. Assume now we are interested in a given abstract analysis, whose domain is $\mathcal{D}$ and whose abstraction map is $\alpha : \mathcal{B} \mapsto \mathcal{D}$. It can easily be seen that a success condition for c is the constraint

$$s(c) = \bigvee_{\models S \wedge c} \alpha(S) \ . \tag{1}$$

The previous formula should be read as follows. We look for the most precise condition which is satisfied by every constraint store S which is consistent with c. Dually, a failure condition for c is the constraint

$$f(c) = \bigvee_{\not\models S \wedge c} \alpha(S) \ . \tag{2}$$

Hence we abstract a concrete constraint into the pair $\left[\begin{smallmatrix} s(c) \\ f(c) \end{smallmatrix}\right]$.

In the case of kernel constraints, we need also to know what happens (from an abstract point of view) if a kernel constraint k is satisfied; this information is obviously $\alpha(k)$. Hence a kernel constraint will be abstracted into $\left[\begin{smallmatrix} s(k) \\ \alpha(k) \\ f(k) \end{smallmatrix}\right]$. This leads to a straightforward definition for $(\propto obs)^\sharp$, i.e. the abstract counterpart of $\propto obs$, $\left[\begin{smallmatrix} s \\ a \\ f \end{smallmatrix}\right] \propto obs^\sharp = \left[\begin{smallmatrix} s \\ f \end{smallmatrix}\right]$.

The conjunction of kernel and observability is defined as

$$\left[\begin{smallmatrix} s_1 \\ a_1 \\ f_1 \end{smallmatrix}\right] \sqcap^\sharp \left[\begin{smallmatrix} s_2 \\ a_2 \\ f_2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} s_1 \wedge s_2 \\ a_1 \wedge a_2 \\ f_1 \vee f_2 \end{smallmatrix}\right] \quad \text{and} \quad \left[\begin{smallmatrix} s_1 \\ f_1 \end{smallmatrix}\right] \sqcap^\sharp \left[\begin{smallmatrix} s_2 \\ f_2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} s_1 \wedge s_2 \\ f_1 \vee f_2 \end{smallmatrix}\right] \ .$$

The disjunction of kernel and observability is dually defined as

$$\left[\begin{smallmatrix} s_1 \\ a_1 \\ f_1 \end{smallmatrix}\right] \sqcup^\sharp \left[\begin{smallmatrix} s_2 \\ a_2 \\ f_2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} s_1 \vee s_2 \\ a_1 \vee a_2 \\ f_1 \wedge f_2 \end{smallmatrix}\right] \quad \text{and} \quad \left[\begin{smallmatrix} s_1 \\ f_1 \end{smallmatrix}\right] \sqcup^\sharp \left[\begin{smallmatrix} s_2 \\ f_2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} s_1 \vee s_2 \\ f_1 \wedge f_2 \end{smallmatrix}\right]$$

Abstract negation and cylindrification are defined as

$$-^\sharp \left[\begin{smallmatrix} s \\ f \end{smallmatrix}\right] = \left[\begin{smallmatrix} f \\ s \end{smallmatrix}\right] \quad \text{and} \quad \exists^\sharp_x \left[\begin{smallmatrix} s \\ f \end{smallmatrix}\right] = \left[\begin{smallmatrix} \exists_x s \\ \exists_x f \end{smallmatrix}\right] \quad \text{and} \quad \exists^\sharp_x \left[\begin{smallmatrix} s \\ a \\ f \end{smallmatrix}\right] = \left[\begin{smallmatrix} \exists_x s \\ \exists_x a \\ \exists_x f \end{smallmatrix}\right] \ .$$

With the above definitions, and by assuming $[\![c]\!]^\sharp = \langle \left[\begin{smallmatrix} \top \\ \_ \end{smallmatrix}\right] + \left[\begin{smallmatrix} s(c) \\ \alpha(c) \\ f(c) \end{smallmatrix}\right] \rangle$, the semantic scheme of definition 3 can be directly translated into an abstract semantics definition. The abstract semantics is defined as $\mathcal{S}^\sharp(P) = \bigsqcup_{i \geq 0} (T^\sharp_P)^i(I^0)$.

Finally we define an abstract version of the $\mathcal{O}$ function as $\mathcal{O}^\sharp(l^\sharp)[d] = \bigvee_{\left[\begin{smallmatrix} s \\ f \end{smallmatrix}\right] + \left[\begin{smallmatrix} s' \\ d \\ f' \end{smallmatrix}\right] \in l^\sharp} s \vee \bigvee_{\left[\begin{smallmatrix} s \\ f \end{smallmatrix}\right] + \left[\begin{smallmatrix} s' \\ d \\ f' \end{smallmatrix}\right] \in l^\sharp} s.$

For a call patterns version of the abstract semantics, one should only perform the same changes suggested for the concrete call patterns semantics. For an example of computation of this semantics, see later.

9 Towards a finitely computable abstract semantics

In the case of abstract analysis of pure logic programs, if the abstract domain is finite or noetherian (or at the least the set of abstract constraints on a finite set of variables is finite), then the abstract analysis is finite. This is because a semantics for pure logic programs uses as computational domain sets of constraints, rather than sequences as in our approach. This problem was already tackled in [13]. As already noted, the semantics in [13] was quite complex, since cut and divergence conditions were "declared" rather than "applied" (as we do in this paper). This in turn led to complex and incomplete "reduction rules" on sequences. The present approach leads to a simple solution. We will show that it is safe to remove a constraint from a sequence if it entails another constraint of the same type and precedes it in the sequence. As a consequence it is not possible to have multiple copies of a constraint in a sequence and therefore the abstract domain of sequences becomes finite. Finally, the abstract semantics is finitely computable.

The reduction rule on sequences can be viewed as a further abstract interpretation process, such that every abstract sequence is abstracted into another abstract sequence where all the "entailing" constraints are removed. Formally, we define an abstraction map as: $\alpha(\langle\rangle) = \langle\rangle$, $\alpha(\langle v\rangle) = v$ and $\alpha(s_1 :: s_2) = \alpha(s_1) :: \alpha'(s_2)$, where $\alpha'(s_2)$ is $\alpha(s_2)$ deprived of all constraints v such that there exists a constraint $w \in \alpha(s_1)$ of the same type as v such that $|v|_1 \sqcup^\sharp |w|_1 = |w|_1$ and $|v|_2 \sqcup^\sharp |w|_2 = |w|_2$, where the expression "of the same type of" means that the two (abstract) constraints should be both closed convergent, or both open convergent and so on. Intuitively, if a constraint is preceded by another constraint which entails it, it is not useful for computing the observable properties we are interested in. Moreover, it will not be useful, even if we combine the sequence in any possible compositional context, as it will be shown in the following paragraphs.

Definition 17 *Let $\mathcal{RSS}$ be the set of reduced sequences, that is the set of sequences such that no constraint precedes another constraint of the same type and which entails it. Obviously, we have $\alpha : \mathcal{SS} \mapsto \mathcal{RSS}$. The following abstract operators are defined on $\mathcal{RSS}$:*

$$\propto obs^a(c) = \propto obs^\sharp(c) \qquad [\![c]\!]^a = \langle \begin{bmatrix} \top \\ \_ \end{bmatrix} + \begin{bmatrix} s(c) \\ \alpha(c) \\ d(c) \end{bmatrix} \rangle$$

$$s_1^a \oplus^a s_2^a = \alpha(s_1^a \oplus^\sharp s_2^a) \qquad s_1^a \otimes^a s_2^a = \alpha(s_1^a \otimes^\sharp s_2^a)$$

$$\exists_x^a s^a = \alpha(\exists_x^\sharp s^a) \qquad !^a(s^a) = \alpha(!^\sharp(s^a))$$

$$¡^a(s^a) = \alpha(¡^\sharp(s^a)) \qquad \mathcal{O}^a(s^a) = \mathcal{O}^\sharp(s^a)\ .$$

The above definition induces a semantic scheme based on the one described in definition 3. We define the abstract reduced semantics as $\mathcal{S}^a(P) = \bigsqcup_{i\geq 0}(T_P^a)^i(I^0)$. The local correctness condition holds: given an interpretation I, we have $\alpha(T_P^\sharp(I)) = T_P^a(\alpha(I))$ where, by definition, $\alpha(I)[\mathtt{p}] = \alpha(I[\mathtt{p}])$.

If v entails w then the success condition of v is lower than the success condition of w and similarly the abstract approximation of v is lower than the abstract approximation of w. Therefore $\mathcal{O}^a(\langle w\rangle) = \mathcal{O}^a(\langle v, w\rangle)$. This allows us to conclude that $\mathcal{O}^a(\alpha(s)) = \mathcal{O}^a(s)$. By using this result it can be shown that $\mathcal{O}^\sharp(\mathcal{S}^\sharp(P)) = \bigvee_{i\geq 0} \mathcal{O}^a\left((T_P^a)^i(\alpha(I^0))\right)$ Note that, by assuming the set of abstract constraints on a finite set of variables to be finite, we conclude that the right hand side of the above equality can be computed in a finite number of steps. A similar reduction can be applied to the call patterns analysis described in section 7 (see the second example below).

10 Examples

In this section we show some examples of the computation of the abstract semantics for simple programs. Consider for instance the following program, which shows the incompleteness of the Prolog search rule

$$\begin{array}{l} \mathtt{eq(X,Y)} : -\mathtt{X} = \mathtt{a}, \mathtt{Y} = \mathtt{b}. \\ \mathtt{eq(X,Y)} : -\mathtt{eq(Y,X)}. \\ \mathtt{eq(X,Y)} : -\mathtt{eq(X,Z)\ and\ eq(Z,Y)}. \\ \mathtt{eq(X,Y)} : -\mathtt{X} = \mathtt{Y}. \end{array}$$

We compute a sharing analysis for computed answer substitutions. We use the classical domain for sharing analysis proposed in [9], but we assume an abstract element $-$ which

represents the empty set of substitutions. Moreover, $\top$ will denote the top of the domain, that is the power set of the set of variables. The abstract computation through $T_P^\sharp$ proceeds as follows (α_1 and α_2 are the first and second parameter of the procedure).

$$
\begin{aligned}
I^0[\mathtt{eq}] &= \langle \widetilde{\left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\end{smallmatrix}\right]} \rangle \\
I^1[\mathtt{eq}] &= \langle \left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \emptyset\end{smallmatrix}\right], \widetilde{\left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\end{smallmatrix}\right]}, \widetilde{\left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\end{smallmatrix}\right]}, \left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ (\alpha_1,\alpha_2)\\ \top\end{smallmatrix}\right] \rangle \\
I^2[\mathtt{eq}] &= \langle \left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \emptyset\end{smallmatrix}\right], \left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \emptyset\end{smallmatrix}\right], \widetilde{\left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\end{smallmatrix}\right]}, \widetilde{\left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\end{smallmatrix}\right]}, \left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ (\alpha_1,\alpha_2)\\ \top\end{smallmatrix}\right], \left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \emptyset\end{smallmatrix}\right], \\
&\quad \widetilde{\left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \emptyset\end{smallmatrix}\right]}, \widetilde{\left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \emptyset\end{smallmatrix}\right]}, \left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \emptyset\end{smallmatrix}\right], \widetilde{\left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\end{smallmatrix}\right]}, \widetilde{\left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\end{smallmatrix}\right]}, \left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \emptyset\end{smallmatrix}\right], \\
&\quad \widetilde{\left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\end{smallmatrix}\right]}, \widetilde{\left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\end{smallmatrix}\right]}, \left[\begin{smallmatrix}-\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ (\alpha_1,\alpha_2)\\ \top\end{smallmatrix}\right] \rangle \\
I^3[\mathtt{eq}] &= \ldots
\end{aligned}
$$

Note that the computation of the abstract fixpoint is not performed in a finite number of steps. However, if we reduce sequences or, equivalently, if we compute the abstract semantics through the T_P^a operator, the abstract fixpoint is reached at step 1. The observability map $\mathcal{O}^a$ tells us that $\mathcal{O}^a(I^1[\mathtt{eq}])[\emptyset] = \top$, that is "the answer where both X and Y are ground is a Prolog computed answer, only if the constraint store satisfies $\top$", which is always true. More interestingly, $\mathcal{O}^a(I^1[\mathtt{eq}])[(\alpha_1, \alpha_2)] = -$ that is, in order to observe a computed answer substitution where X and Y share, we must start the computation from a constraint store where $-$ is satisfied, which can never be the case. Hence we conclude that X and Y can never share after the execution of the `eq` procedure. Note that this information cannot be derived through an abstract analysis based on a logic programming interpretation of Prolog.

The following example shows the computation of the abstract call patterns semantics in a simple case. Now we use an abstract domain able to model groundness and non–freeness, without directionality. Even this "weak" domain is able to show the usefulness of our approach. Consider the Prolog program

$$\mathtt{p(X)} : -\mathtt{X} = 4, !. \quad \mathtt{p(X)} : -\mathtt{q(X)}. \quad \mathtt{q(X)} : -\mathtt{X} = 5. \quad \mathtt{q(X)} : -\mathtt{p(X)}.$$

which is translated into our abstract syntax ...

$$\mathtt{p(X)} : -\mathtt{cut(X} = 4) \ \mathtt{or}\ \mathtt{q(X)}. \qquad\qquad \mathtt{q(X)} : -\mathtt{X} = 5 \ \mathtt{or}\ \mathtt{p(X)}. \ ,$$

... and then abstractly compiled into:

$$\mathtt{p(X)} : -\mathtt{cut}(\left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ g(\mathtt{X})\\ nf(\mathtt{X})\end{smallmatrix}\right]) \ \mathtt{or}\ \mathtt{q(X)}. \qquad\qquad \mathtt{q(X)} : -\left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ g(\mathtt{X})\\ nf(\mathtt{X})\end{smallmatrix}\right] \ \mathtt{or}\ \mathtt{p(X)}.$$

The computation of the abstract fixpoint (with sequence reductions) stops at the second iteration in such a way that

$$lfp(T_P)[\mathtt{p}] = \langle \overline{\left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\\ \_\end{smallmatrix}\right]; \mathtt{p}}, \left[\begin{smallmatrix}\top\\ \_\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ g(\alpha)\\ nf(\alpha)\end{smallmatrix}\right], \overline{\left[\begin{smallmatrix}nf(\alpha)\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}\top\\ \top\\ \_\end{smallmatrix}\right]; \mathtt{q}}, \widetilde{\left[\begin{smallmatrix}nf(\alpha)\\ \top\end{smallmatrix}\right] + \left[\begin{smallmatrix}nf(\alpha)\\ \top\\ \_\end{smallmatrix}\right]; \mathtt{p}} \rangle \ .$$

From the third constraint of this denotation we conclude that the unique call pattern for `q`, which can arise from the execution of `p`, is observable only in a constraint store in which α (or, equivalently, X) is a non free variable. This information can be used to optimize the compilation of the unification for the clauses of the definition of `q`. Note that it would not have been possible to determine this information if we had discarded the cut operator in our analysis.

11 Conclusions and future work

This paper shows a general approach to a precise abstract interpretation of Prolog programs. Note that our approach deals naturally with negation as finite failure, which can easily be implemented through a clever use of the cut operator. Simple extensions of our approach can deal with several Prolog built–in's and even with error conditions. The computation of the

abstract analysis is finite, which was not the case in our previous proposal [13]. The overhead needed for reaching an improved precision essentially consists in the use of observability and kernel constraints and of their success and failure conditions.

We can now consider the already mentioned paper [11]. A big difference arises looking for instance at the way they handle clause composition and the cut. Their definitions are the following ones.

> Consider two sequences S_1 and S_2 without cut information. S_1 stands for the result of c_i and S_2 stands for the (combined) result of $c_{i+1}, \ldots, c_n$. If execution of c_i terminates, then suffix $c_{i+1}, \ldots, c_n$ is executed. Otherwise $c_{i+1}, \ldots, c_n$ is not executed. Therefore, the combined result $S_1 \Box S_2 = S_1 :: S_2$ if S_1 is finite (i.e. neither incomplete nor infinite) and $S_1 \Box S_2 = S_1$ otherwise. The definition can be extended to sequences with cut information. If no cut is executed in c_i (...), the previous reasoning applies. Otherwise, suffix $c_{i+1}, \ldots, c_n$ is not executed (...). So, the combined result (...) is defined by $\langle S_1, cf \rangle \Box S_2 = S_1 \Box S_2$ if $cf = nocut$ and $\langle S_1, cf \rangle \Box S_2 = S_1$ if $cf = cut$.

Compare this with our definitions 10 and 11. The difference is that we delay the decision divergence/no divergence or cut/no cut to the actual evaluation of the denotation in a given constraint store. They take instead this decision just when they compose the denotations. It is not clear how their approach could lead to a goal-independent denotation. In any case their approach makes impossible to compute the denotations of the most general goals only, like we do in this paper. This is important because it leads to a simpler and more efficient computation of the semantics.

While a general approach for defining success and failure conditions is available (equations 1 and 2), one should not think that the problem of finding success and failure conditions is definitely solved. Actually, we only shifted the problem in the definition of the abstract domain, which must be devised in such a way that equations 1 and 2 do not give trivial approximations (like $\top$, for instance). Future work will be spent in devising such domains, and in showing the feasibility of the approach with its implementation.

References

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

[2] R. Barbuti, M. Codish, R. Giacobazzi, and M. Maher. Oracle Semantics for PROLOG. *Information and Computation*, 1995.

[3] E. Börger. A Logical Operational Semantics of Full Prolog. In E. Börger, H. Kleine, H. Büning, and M. Richter, editors, *CSL 89. 3rd workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*, pages 36–64. Springer-Verlag, 1990.

[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. The MIT Press, 1993.

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

[6] A. de Bruin and E. de Vink. Continuation Semantics for Prolog with Cut. In J. Diaz and F. Orejas, editors, *Proc. CAAP 89*, volume 351 of *Lecture Notes in Computer Science*, pages 178–192. Springer-Verlag, 1989.

[7] S. K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming*, 5:61–91, 1988.

[8] Melvin Fitting. A Deterministic Prolog Fixpoint Semantics. *Journal of Logic Programming*, 2(2):111–118, 1985.

[9] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992.

[10] N. D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for PROLOG. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 281–288, 1984.

[11] 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, 1994.

[12] 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*, pages 521–532, 1995.

[13] G. Levi and F. Spoto. Accurate Analysis of Prolog with Cut. In P. Lucio, M. Martelli, and M. Navarro, editors, *Proceeding APPIA-GULP-PRODE'96*, pages 481–492, 1996.