A Fixed Point Semantics for Logic Programs Extended with Cuts

Wim Vanhoof, Remko Tronçon, and Maurice Bruynooghe

Department of Computer Science, K.U.Leuven, Belgium
{wimvh,remko,maurice}@cs.kuleuven.ac.be

Abstract. In this paper, we develop a bottom-up fixed point semantics for pure Prolog programs extended with !/0 that allows to reconstruct the operational semantics of a particular goal. Our semantics captures both the order in which solutions are computed by SLD-resolution and their multiplicity.

1 Introduction

Semantics of logic programming is an active research area within logic programming, and several approaches have been developed within the field to define the "meaning" of a program. The ability to define the meaning of a program in a formal way has a number of clear advantages. It cannot only help understanding the program, but it also provides a formal way to prove correctness of program transformations (like e.g. partial evaluation [14]). Furthermore, abstracting a suitable semantics is an appealing way to derive correct program analyses or transformations. One of the most attractive features of logic programming is the equivalence between the so-called *declarative* (or fixed point) semantics of a logic program and its *operational* semantics. In its most basic form, the declarative semantics of a logic program P is defined as its minimal Herbrand model, which can be shown equivalent [26] to the success set of the program, when the latter is defined as the set of ground atoms that can be refuted in P.

The s-semantics approach [12, 5] to logic programming extends this basic equivalence result by allowing variables in the Herbrand universe. The resulting fixed point semantics computes a denotation of a program P that, when combined with an atomic query, characterises exactly the set of computed answer substitutions of that query in P. By abstracting the fixed-point semantics, one can obtain a program analysis that computes goal-independent analysis results that correctly model the computed answer substitutions created when constructing refutations. This makes the s-semantics an appealing semantics for program analysis and transformation and it has been used as a foundation of several abstract interpretation frameworks [3, 9].

Appealing as the s-semantics approach may be for logic program analysis and transformation, it does not model any control information that is associated with evaluating a Prolog program. While the s-semantics correctly models the computed answer substitutions that are computed by SLD-resolution, it does

not model the order in which these answers are computed, nor their multiplicity. Consequently, the s-semantics is not a sufficient basis for the analysis or transformation of Prolog programs when order and multiplicity of the computed answers need to be preserved. These control aspects are important when analysing Prolog programs, in particular in the presence of cuts.

In this work, we develop an alternative fixed point semantics. The bottom-up derivations in the resulting denotation incorporate the presence of cuts in the program and contain the necessary control information such that a simple combination operator suffices to retrieve a sequence of answers for a particular atomic query. The resulting sequence is equivalent with the sequence of computed answers as returned by SLD-resolution using the left-to-right selection rule of Prolog. Our semantics hence captures the order in which solutions are computed, their multiplicity and deals with programs that contain cut constructs. The main motivation for our work is to construct a clean and simple fixed point semantics that can easily be abstracted to obtain goal-independent program analyses or transformations that preserve the operational semantics of the program under SLD-resolution. In particular, we feel our semantics can serve as a semantic basis for bottom-up partial evaluation.

The remainder of this paper is organised as follows. In Section 2 we introduce some basic concepts and notation, and define the operational semantics of a program as a sequence of computed answers. In Section 3 we introduce a fixed point semantics for pure Prolog programs and prove the equivalence between this semantics and the operational semantics defined in Section 2. We extend these results towards programs that contain the cut operator in Section 4. We conclude the paper in Section 5, where we discuss related work and give a possible application of our semantics.

2 Preliminaries

In what follows, we assume the reader to be familiar with the basic logic programming concepts as they are found, for example, in [1, 18]. In this work, we restrict ourselves to definite programs, and assume the Prolog variant of logic programming. In particular, we consider a program to be defined as a sequence of program clauses, where each program clause is labelled by a natural number identifying its position in the program. We will denote the i-th program clause as $H \overset{i}{\leftarrow} B_1, \ldots, B_n$, although the label may be dropped when irrelevant. As usual, a substitution is defined as a finite mapping from distinct variables to terms. The set of all substitutions is denoted by $Subst$. As usual, we denote substitutions as $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ where each $X_i \neq t_i$. Given substitutions θ and σ, $\theta\sigma$ denotes there composition and if V is a set of variables, then $\theta_{|_V}$ denotes θ restricted to V. The substitution ϵ denotes the empty substitution. If E is an expression (be it a term, atom or clause) and θ a substitution, then $E\theta$ denotes the result of applying θ to E and is defined as the expression obtained from E by simultaneously replacing the variables from the domain of θ that occur in E by their corresponding term. We call $E\theta$ an instance of E. If E is an expression

and F is an instance of E, then E is said to be more general than F, denoted $E \preceq F$, or simply a generalisation of F. If E and F are expressions such that E is an instance of F and F is an instance of E, then E and F are called variants, denoted by $E \approx F$. A substitution θ such that $E\theta \approx E$ is called a renaming substitution for E. Given a set of expressions I and a syntactic object C, we denote by $A_1, \ldots, A_n \ll_C I$ that $A_1, \ldots, A_n$ are variants of expressions in I renamed apart from C and from each other. If $E \preceq F$ and $E \not\approx F$, we say that E is strictly more general than F, denoted with $E \prec F$. A substitution θ is a unifier for atoms A and B if $A\theta = B\theta$; it is a most general unifier for A and B if for each unifier σ of A and B there exists a substitution γ such that $\sigma = \theta\gamma$. As usual, we denote the most general unifier for atoms A and B (modulo variable renaming) by $mgu(A, B)$. The notion of mgu is easily extended to conjunctions of atoms. As usual, we assume a goal (or query) to be a clause of the form $\leftarrow A_1, \ldots, A_n$. In the rest of the paper, we will often denote such a goal by the symbol Q.

The operational behavior of a logic program is characterised by the notion of SLD-derivation [1, 18]. As we have introduced labeled clauses, we can use labels instead of clauses when formalising the notion of SLD-derivation. Hence, an SLD-derivation of a goal Q_0 in a program P consists of a possibly infinite sequence of goals, denoted $Q_0 \overset{i_1,\theta_1}{\rightarrow} Q_1 \overset{i_2,\theta_2}{\rightarrow} Q_2 \ldots$ such that each Q_j is derived from Q_{j-1} and a freshly renamed copy of the i_j-th program clause using the most general unifier θ_j. Since we consider the Prolog execution mechanism, we assume that SLD-derivations are constructed using the left-to-right selection rule, that is: a goal Q_{j+1} is derived from Q_j and a clause C using θ if and only if Q_{j+1} is the goal $\leftarrow (B_1, \ldots, B_n, A_2, \ldots, A_m)\theta$ where $Q_j = \leftarrow A_1, \ldots, A_m$, C is the clause $H \leftarrow B_1, \ldots, B_n$ and $\theta = mgu(H, A_1)$. An SLD-derivation can be finite or infinite. A finite SLD-derivation that ends in an empty goal (i.e. a goal without any atom) is called a successful derivation, or an SLD-refutation. We denote an SLD-refutation of Q in P by $Q \rightsquigarrow_P \Box$. A finite derivation that ends in a goal of which the selected atom does not unify with the head of any program clause is called a failed SLD-derivation. A substitution θ is a computed answer substitution (abbreviated as *cas*) for Q in P if there exists an SLD-refutation $Q \rightsquigarrow_P \Box$ such that σ is the composition of the most general unifiers associated to the derivation and θ is the restriction of σ to the variables of Q. We denote this fact by $Q \overset{\theta}{\rightsquigarrow}_P \Box$. In what follows we also use partial derivations and write $Q \overset{\theta}{\rightsquigarrow}_P R$ with θ the restriction to the variables of Q of the composition of the mgus (θ is called the cas of the partial derivation).

Since we assume that SLD-derivations are constructed using the left-to-right selection rule, such a derivation and its associated computed answer substitution are completely characterised (modulo variable renaming) by the sequence of labels identifying the clauses that were used in the derivation.

Definition 1 (Characteristic derivation). *The* characteristic derivation *of the SLD-derivation*

$$Q_0 \overset{i_1,\theta_1}{\rightarrow} Q_1 \overset{i_2,\theta_2}{\rightarrow} \ldots$$

is the sequence $\langle i_1, i_2, \ldots \rangle$ *of natural numbers.*

Being sequences of natural numbers, we can impose the lexicographic order relation on characteristic derivations: $\langle i_1, i_2, \ldots\rangle > \langle j_1, j_2, \ldots\rangle$ if and only if either $i_1 > i_2$ or $i_1 = j_2$ and $\langle i_2, \ldots\rangle > \langle j_2, \ldots\rangle$. Note that this order relation induces an order relation over SLD-refutations. In what follows, we will often write $Q_1 \rightsquigarrow \Box > Q_2 \rightsquigarrow \Box$ if the characteristic derivation of $Q_1 \rightsquigarrow \Box$ is larger then the characteristic derivation of $Q_2 \rightsquigarrow \Box$. Note also that the order between characteristic refutations for a goal Q reflects precisely the order in which the refutations are constructed and answers are returned by a system that uses a left-to-right selection rule in combination with lexicographic ordering, as Prolog systems do.

Example 1. Consider as a working example the program in Fig. 1, consisting of the definition of the `path/2` predicate and a database of facts `edge/2`. The clauses are labelled for later reference. One can construct the following refutations of

```
(1) path(X,Y):- edge(X,Y).                (3) edge(a,b).
(2) path(X,Y):- edge(X,Z), path(Z,Y).     (4) edge(a,c).
                                          (5) edge(c,b).
```

Fig. 1.

$\leftarrow path(a, X)$.

$$
\begin{aligned}
(\leftarrow path(a,X)) &\overset{1}{\rightarrow} (\leftarrow edge(a,X)) \overset{3,X=b}{\rightarrow} \Box \\
(\leftarrow path(a,X)) &\overset{1}{\rightarrow} (\leftarrow edge(a,X)) \overset{4,X=c}{\rightarrow} \Box \\
(\leftarrow path(a,X)) &\overset{2}{\rightarrow} (\leftarrow edge(a,Z), path(Z,X)) \overset{4,Z=c}{\rightarrow} \\
&\qquad (\leftarrow path(c,X)) \overset{1}{\rightarrow} (\leftarrow edge(c,X)) \overset{5,X=b}{\rightarrow} \Box
\end{aligned}
$$

Consequently, the characteristic derivations of these refutations are respectively $(1,3)$, $(1,4)$ and $(2,4,1,5)$ and we have that $(2,4,1,5) > (1,4) > (1,3)$.

The lexicographic order relation allows to define the semantics behavior of a Prolog program as follows:

Definition 2 (Operational semantics). *Let P be a program and Q a goal. The* operational behavior *of Q w.r.t. P is defined as $\mathcal{O}(P,Q) = \langle \theta_1, \theta_2, \theta_3, \ldots\rangle$ where*

- $\theta_i \in \mathcal{O}(P,Q)$ *if and only if* $Q \overset{\theta_i}{\rightsquigarrow}_P \Box$.
- $\forall \theta_i, \theta_j \in \mathcal{O}(P,Q)$*: if* $j > i$ *then* $Q \overset{\theta_j}{\rightsquigarrow}_P \Box > Q \overset{\theta_i}{\rightsquigarrow}_P \Box$

Modeling the operational behavior of a logic program as a sequence of answers rather than as a set (as e.g. in [12,5]) explicitly takes into account the order in which solutions are found, and allows for multiple occurrences of the same answers (as e.g. in [4,22,11,8]).

Example 2. If P represents the program in Fig. 1 and Q the goal $\leftarrow path(a, X)$, then we have that $\mathcal{O}(P, Q) = (\{X/b\}, \{X/c\}, \{X/b\})$.

Note that $\mathcal{O}(P, Q)$ may be an infinite sequence, modeling the case where the Prolog evaluation mechanism constructs an infinite number of derivations and returns an infinite number of answers. Also note that an infinite derivation does not contribute to $\mathcal{O}(P, Q)$. Hence the fact that $\theta_i \in \mathcal{O}(P, Q)$ does not imply that Prolog computes it, as Prolog can get lost in an infinite derivation before arriving at the derivation leading to θ_i. However, if θ_i *is* computed, so are $\theta_1, \ldots, \theta_{i-1}$.

In what follows, we define a bottom-up characterisation of the operational semantics. We first define the semantics for pure Prolog programs without cut, and extend it later on to deal with the cut operator.

3 A Bottom-up Semantics for Pure Prolog

The s-semantics approach [12, 5] defines a fixed point semantics that captures to some extent the procedural behavior of logic programs. The usual Herbrand base [18, 1] is extended to the set of all the possibly non-ground atoms modulo variance and an immediate consequences operator, say T_P^s, is defined. Given a set of atoms I, $T_P^s(I)$ comprises a new set of atoms that can be derived by unifying atoms from I with the body atoms of a clause in P. The fixed point semantics of a program then equals the fixed point of the T_P^s-operator on an initially empty set of atoms, denoted by $\mathrm{lfp}(T_P^s)$. These notions are due to [12, 5, 6]. This semantics captures the operational behavior of a program to the extent that $p(X_1, \ldots, X_n)\theta \in \mathrm{lfp}(T_P^s)$ if and only if $p(X_1, \ldots, X_n) \overset{\theta}{\leadsto}_P \square$ [12, 5]. As said before, it does not capture the order in which solutions are computed by SLD-resolution, nor their multiplicity. If we want to model the operational behavior of a program as defined above by a bottom-up, fixed point semantics, the atoms in $\mathrm{lfp}(T_P^s)$ need to be related to each other in the same way that the corresponding SLD-derivations are related. In what follows, we introduce an operator that employs the labels that are associated to the program clauses in order to trace *how* an atom was derived by bottom-up derivation. Given this trace, it becomes possible to reconstruct the characteristic derivation of an SLD-refutation for the atom, as such providing a suitable mechanism for relating the atoms derived bottom-up.

In order to trace how an atom is derived by an immediate consequences operator, we associate each such atom with an AND-tree whose nodes are labelled with the labels identifying the clauses that were used for resolution. That is, if a clause $H \overset{l}{\leftarrow} B_1, \ldots, B_n$ is used to derive some atom A from atoms $A_1, \ldots, A_n$ then the AND-tree associated with A has root l and has as children the AND-trees τ_i associated with the derivations of the atoms A_i. In what follows, we use the notation $l(\tau_1, \ldots, \tau_n)$ for an AND-tree with label l and children $\tau_1, \ldots, \tau_n$. We call the combination of an atom with such an AND-tree a *bottom-up derivation*. If we denote the set of all bottom-up derivations by $\mathcal{BU}$, we can define our immediate consequences operator T_P^d as follows:

Definition 3 (T_P^d-operator). *Let P be a definite program and I a set of bottom-up derivations. We define $T_P^d : \wp(\mathcal{BU}) \mapsto \wp(\mathcal{BU})$ as follows:*

$$T_P^d(I) = \left\{ (H\theta, \tau) \middle| \begin{array}{l} \textit{Let } C \textit{ be the clause } H \overset{l}{\leftarrow} B_1, \ldots, B_n \in P, \\ (A_1, \tau_1), \ldots, (A_n, \tau_n) \ll_C I, \\ \theta = mgu((A_1, \ldots, A_n), (B_1, \ldots, B_n)), \\ \tau = l(\tau_1, \ldots, \tau_n) \end{array} \right\}$$

T_P^d is monotonic on the complete lattice $(\wp(\mathcal{BU}), \subseteq)$ and hence the existence of its least fixed point, $\mathrm{lfp}(T_P^d)$ is immediate [18]. The fixed point semantics of a program P is then defined as $\mathcal{F}(P) = \mathrm{lfp}(T_P^d)$. For an atom A, we define $\mathcal{F}_A(P)$ as the set of bottom-up derivations from $\mathrm{lfp}(T_P^d)$ of which the atom unifies with A, formally:

Definition 4. *Let P be a definite program and A an atom. We define*

$$\mathcal{F}_A(P) = \{(A', \tau) \mid (A', \tau) \in \textit{lfp}(T_P^d) \textit{ and } mgu(A, A') \textit{ exists.}\}.$$

Note that it is possible for a single atom A' to occur multiple times in $\mathcal{F}_A(P)$, each time with a different associated bottom-up derivation. This corresponds with $\mathcal{O}(P, \leftarrow A)$ in the fact that a single answer can be returned multiple times, each time with a different computation. This is in contrast with the classic T_P operator, where the lack of the explicit derivations prohibits multiple occurrences of the same atom in $\mathrm{lfp}(T_P)$.

Example 3. Let P denote the program in Fig. 1. We have that

$$\mathcal{F}_{path(X,Y)}(P) = \left\{ \begin{array}{ll} (path(a,b), 1(3)) & (path(a,c), 1(4)) \\ (path(c,b), 1(5)) & (path(a,b), 2(4, 1(5))) \end{array} \right\}$$

By a standard technique, we get that $(A, \tau) \in \mathrm{lfp}(T_P^d)$ if and only if $\leftarrow A \overset{\epsilon}{\leadsto}_P \square$ with ϵ the empty computed answer substitution. Being selection-rule independent, a bottom-up derivation characterises a number of SLD-refutations; one for each combination of selected atoms in successive SLD-derivation steps. Recall that we want to relate our fixed point semantics with the operational semantics that is defined in terms of SLD-derivations using the left to right selection rule. Hence, we are in particular interested in relating a bottom-up derivation (A, τ) with an SLD-refutation for A that is constructed using the left-to-right selection rule. We show that by collecting the labels of the AND-tree in a bottom-up derivation (A, τ) in a depth-first, left-to-right manner, one obtains the characteristic derivation of an SLD-refutation for A.

Definition 5. *Let τ denote a bottom-up derivation. The* depth-first, left-to-right traversal *of τ, denoted by $\overline{\tau}$ is the sequence over $\mathbb{N}$ recursively defined as follows:*

$$\overline{l(\tau_1, \ldots, \tau_n)} = l \bullet \overline{\tau}_1 \bullet \ldots \bullet \overline{\tau}_n$$

where $\bullet$ denotes concatenation of sequences.

For the bottom-up derivations of Example 1, we have that $\overline{1(3)} = (1,3)$ and $\overline{2(4,1(5))} = (2,4,1,5)$. Note that the order over sequences induces an order over AND-trees. In what follows, we write $\tau_1 < \tau_2$ when $\overline{\tau_1} < \overline{\tau_2}$. Now, the following property holds:

Proposition 1. *Let P be a program. If $(A,\tau) \in lfp(T_P^d)$ and $\overline{\tau}$ denotes the depth-first, left-to-right traversal of τ, then $\overline{\tau}$ is the characteristic derivation of an SLD-refutation of $\leftarrow A$ in P with empty computed answer substitution, constructed by the left-to-right selection rule.*

Proof. The proof is by induction on the number of iterations of T_P^d. First, assume $(A,\tau) \in T_P^d \uparrow 1$. Since $(A,\tau) \in T_P^d(\emptyset)$, τ must be of the form $l()$ (having no children) which means that $A \overset{l}{\leftarrow}$ is a clause in P. Consequently, there exists an SLD-refutation of $\leftarrow A$ in P with cas ϵ whose characteristic derivation is l.

Now, assume that it holds for bottom-up derivations obtained in k iterations (induction hypothesis). Assume that $(A,\tau) \in T_P^d \uparrow k+1$. By construction, τ is of the form $l(\tau_1,\ldots,\tau_n)$ and $\overline{\tau} = l \bullet \overline{\tau_1} \bullet \ldots \bullet \overline{\tau_n}$. By definition, $A = H\theta$ with $H \overset{l}{\leftarrow} B_1,\ldots,B_n \in P$, $\theta = mgu((A_1,\ldots,A_n),(B_1,\ldots,B_n))$ and $(A_1,\tau_1),\ldots,(A_n,\tau_n) \in T_P^d \uparrow k$. Since $A = H\theta$, we have that $mgu(H,A) = \theta_{|_H}$ and hence there exists a partial SLD-derivation $\leftarrow A \overset{\epsilon}{\leadsto}_P (\leftarrow B_1,\ldots,B_n)\theta_{|_H}$ with the sequence l as its characteristic derivation. Note that the cas is ϵ as A is an instance of H. Now, remains to prove that $(\leftarrow B_1,\ldots,B_n)\theta_{|_H} \overset{\sigma}{\leadsto}_P \square$ with $\overline{\tau}_1 \bullet \ldots \bullet \overline{\tau}_n$ as the characteristic derivation and σ restricted to the variables of $A(= H\theta_{|_H})$ the empty substitution, i.e. $dom(\sigma) \cap vars(H\theta_{|_H}) = \emptyset$.

We can write θ as $\theta_{|_H} \cup \theta_B$ with $dom(\theta_B) \cap vars(H\theta_{|_H}) = \emptyset$. By induction hypothesis, for each i, $\leftarrow A_i$ has a refutation with cas ϵ and characteristic derivation $\overline{\tau}_i$. Hence $\leftarrow A_1,\ldots,A_n$ has a refutation with cas ϵ and characteristic derivation $\overline{\tau}_1 \bullet \ldots \bullet \overline{\tau}_n$. Hence also $(\leftarrow A_1,\ldots,A_n)\theta$ has a refutation with cas ϵ and characteristic derivation $\overline{\tau}_1 \bullet \ldots \bullet \overline{\tau}_n$. But $(\leftarrow A_1,\ldots,A_n)\theta = (\leftarrow B_1,\ldots,B_n)\theta = (\leftarrow B_1,\ldots,B_n)(\theta_{|_H} \cup \theta_B) = (\leftarrow B_1,\ldots,B_n)\theta_{|_H}\theta_B$. Let μ be the substitution accumulated in the refutation of $(B_1,\ldots,B_n)\theta_{|_H}\theta_B$. Note that $dom(\mu) \cap varsH\theta = \emptyset$ because $(dom(\theta_B) \cup dom(\mu)) \cap vars(H\theta) = \emptyset$ and hence also $dom(\theta_B\mu) \cap varsH\theta = \emptyset$. According to the lifting lemma [18], there exists a refutation for $(\leftarrow B_1,\ldots,B_n)\theta_{|_H}$ with accumulated substitution μ' such that $\theta_B\mu = \mu'\gamma$ for some γ. We have $dom(\mu') \subseteq dom(\theta_B\mu)$ hence $dom(\mu') \cap vars(H\theta_{|_H}) = \emptyset$; this holds also for the cas σ of the refutation with accumulated substitution μ'. $\square$

The above property relates the bottom-up derivation (A,τ) with an SLD-refutation (constructed using the left-to-right selection rule) for $\leftarrow A$ and provides as such the basis to relate the fixed point semantics with the operational semantics. Observe, however that the reverse of Proposition 1 does not necessarily hold. An atom A might have a refutation in P with an empty answer substitution, $\leftarrow A \overset{\epsilon}{\leadsto}_P \square$ while $\text{lfp}(T_P^d)$ does not contain A but only a generalisation of

A. However, if we restrict our attention to atomic goals of the form $p(\overline{X})$, i.e. all arguments variables, the following does hold[1].

Lemma 1. *Let P be a definite program. If $\leftarrow p(\overline{X}) \overset{\theta}{\leadsto}_P \square$ with the characteristic derivation δ, then there exists a substitution σ and a labeled* AND*-tree τ such that $(p(\overline{X})\sigma, \tau) \in lfp(T_P^d)$, $\overline{\tau} = \delta$ and $p(\overline{X})\theta \approx p(\overline{X})\sigma$.*

Proof. We prove this by induction on the length of the SLD-refutation. First, consider a refutation $\leftarrow p(\overline{X}) \overset{\theta}{\leadsto}_P \square$ of length 1, with characteristic derivation the single label l. This means that there exist a renamed clause $p(\overline{X})\theta) \leftarrow \in P$, and consequently we have that a renaming of $(p(\overline{X})\theta, l) \in \mathrm{lfp}(T_P^d)$, i.e. $(p(\overline{X})\sigma, l) \in \mathrm{lfp}(T_P^d)$ with $p(\overline{X})\sigma \approx p(\overline{X})\theta$.

Next, assume that it holds for SLD-refutations of length k. Consider an SLD-refutation $\leftarrow p(\overline{X}) \overset{\theta}{\leadsto}_P \square$ of length $k+1$, with characteristic derivation $l, l_1, \ldots, l_k$. If the clause labelled l is of the form $H \overset{l}{\leftarrow} B_1, \ldots, B_n$, then the first step in the refutation is the derivation $\leftarrow p(\overline{X}) \overset{\sigma}{\leadsto}_P \leftarrow B_1, \ldots, B_n$ with $H = p(\overline{X})\sigma$. To ease the formulation of the proof, let us – without loosing generality – assume that $n = 2$; i.e. $H \overset{l}{\leftarrow} B_1, B_2$. Now, the remainder of the SLD-refutation of $p(\overline{X})$ can be split into the refutations $\leftarrow B_1 \overset{\theta_1}{\leadsto}_P \square$ and $\leftarrow B_2\theta_1 \overset{\theta_2}{\leadsto}_P \square$ with characteristic derivations $l_1, \ldots, l_s$ and $l_{s+1}, \ldots, l_k$ respectively. If we assume that $B_1 = q_1(\overline{Y})\gamma_1$, there also exists an SLD-refutation $\leftarrow q_1(\overline{Y}) \overset{\sigma_1}{\leadsto}_P \square$ such that $\theta_1 = mgu(B_1, q_1(\overline{Y})\sigma_1)$. Likewise, if we assume that $B_2 = q_2(\overline{Z})\gamma_2$, there also exists an SLD-refutation $\leftarrow q_2(\overline{Z}) \overset{\sigma_2}{\leadsto}_P \square$ such that $\theta_2 = mgu(B_2\theta_1, q_2(\overline{Z})\sigma_2)$. Although B_1 and B_2 may have variables in common, $dom(\theta_1) \cap dom(\theta_2) = \emptyset$: no variable can be bound by both θ_1 and θ_2 since $q_1(\overline{Y})\sigma_1$ and $q_2(\overline{Z})\sigma_2$ have no variables in common and if θ_1 binds a variable, every occurrence of that variable is bound in $B_2\theta_1$ and it cannot be in $dom(\theta_2)$. Consequently, we have that $\theta_1\theta_2 = mgu((B_1, B_2), (q_1(\overline{Y})\sigma_1, q_2(\overline{Z})\sigma_2))$ (1).

Now, by induction hypothesis, a renaming of $(q_1(\overline{Y})\sigma_1, \tau_1) \in \mathrm{lfp}(T_P^d)$ and $(q_2(\overline{Z})\sigma_2), \tau_2) \in \mathrm{lfp}(T_P^d)$ with τ_1 and τ_2 such that $\overline{\tau}_1 = l_1, \ldots, l_s$ and $\overline{\tau}_2 = l_{s+1}, \ldots, l_k$. By (1), it follows then that $(H\theta_1\theta_2, l(\tau_1, \tau_2)) \in \mathrm{lfp}(T_P^d)$. We have that $H\theta_1\theta_2 = p(\overline{X})\sigma\theta_1\theta_2 = p(\overline{X})\theta$. Moreover, $\overline{l(\tau_1, \tau_2)} = l, l_1, \ldots, l_s, l_{s+1}, \ldots, l_k$ which concludes the proof. $\square$

Together, Proposition 1 and Lemma 1 extend a classic result (found e.g. in [12, 5]) which essentially states the equivalence between the *set* of computed answers Θ for a goal $\leftarrow p(X_1, \ldots, X_n)$ and the atoms $\{p(X_1, \ldots, X_n)\theta \mid \theta \in \Theta\}$ from the least fixed point of the immediate consequences operator. The following corollary extends this result to regular atoms:

Corollary 1. *Let P be a definite program and A an atom. If $A \overset{\theta}{\leadsto}_P \square$ with characteristic derivation δ, then there exists an atom A' and a labelled* AND*-tree τ such that $(A', \tau) \in lfp(T_P^d)$, $\overline{\tau} = \delta$ and $A\theta = A'\gamma$ with $\gamma = mgu(A', A)$.*

[1] Note that this restriction does not diminish the generality of the technique but eases the formulation of the results.

The equivalence of the depth-first, left-to-right traversal of the AND-tree τ with the characteristic derivation of the atom allows to order the atoms in $\mathcal{F}_{p(\overline{X})}(P)$ lexicographically, such that the order between the atoms $p(\overline{t_1})$, $p(\overline{t_2})$,... reflects the order in which the corresponding computed answers $\{\overline{X}/\overline{t_1}\}$, $\overline{X}/\overline{t_2}\}$,... are returned for the goal $\leftarrow p(\overline{X})$. In what follows, we consider $(A, \tau_A) < (B, \tau_B)$ if and only if $\tau_A < \tau_B$. Note that, by construction, $<$ is a total order relation on the set $\mathrm{lfp}(T_P^d)$. Given $\mathcal{F}_A(P)$, we denote with $\mathcal{F}_A^{\langle\rangle}(P)$ the sequence containing the bottom-up derivations from $\mathcal{F}_A(P)$ sorted using $<$.

Definition 6. *Let P be a program and A an atom. Then we define*

$$\mathcal{F}_A^{\langle\rangle}(P) = \langle (A_1, \tau_1), (A_2, \tau_2), \ldots \rangle$$

where each $(A_i, \tau_i) \in \mathcal{F}_A(P)$ and for each (A_i, τ_i), $(A_j, \tau_j) \in \mathcal{F}_A(P)$: $j > i$ if and only if $(A_j, \tau_j) > (A_i, \tau_i)$.

The following theorem states that the operational semantics of a goal of the form $\leftarrow p(\overline{X})$ is modeled precisely by $\mathcal{F}_{p(\overline{X})}^{\langle\rangle}(P)$, obtained from the program's fixed point semantics $\mathrm{lfp}(T_P^d)$.

Theorem 1. *Let P be a definite program and A an atom of the form $p(\overline{X})$. Assume that $\mathcal{F}_A^{\langle\rangle}(P) = \langle (A_1, \tau_1), (A_2, \tau_2), \ldots \rangle$ and that $\mathcal{O}(P, \leftarrow A) = \langle \sigma_1, \sigma_2, \ldots \rangle$. Then we have that for all i $A_i \approx A\sigma_i$.*

Again, we can generalise this result to regular atoms:

Corollary 2. *Let P be a definite program and A an atom. Assume that $\mathcal{F}_A^{\langle\rangle}(P) = \langle (A_1, \tau_1), (A_2, \tau_2), \ldots \rangle$ and that $\mathcal{O}(P, \leftarrow A) = \langle \sigma_1, \sigma_2, \ldots \rangle$. Then we have that for all i $A_i\theta_i \approx A\sigma_i$ where $\theta_i = mgu(A, A_i)$.*

4 A Bottom-up Semantics for Pure Prolog extended with cut

In what follows, we extend the fixed point semantics defined in the previous section in order to deal with pure Prolog programs that possibly contain !/0 constructs. Programs that use the !/0 construct manipulate the SLD search process by literally cutting away part of the search space, i.e. avoiding the construction of a number of SLD-derivations. If we want to retain the equivalence between the operational and fixed point semantics for programs that contain cuts, we must ensure that $\mathcal{F}_A(P)$ does not contain any answers that would have been cut away during the SLD search process for $\leftarrow A$ in P. Let us first recall the semantics of !/0. Assume that during the construction of an SLD-derivation an atom A is unified with a clause $H \leftarrow B_1, \ldots, !, \ldots, B_n$ containing a !/0. The atom A is called the *parent atom* of the !/0 and the meaning of the !/0 is that it succeeds and discards all alternative derivations that may originate between (and including) the parent atom and the !/0 [18].

```
(1) path(X,Y):- edge(X,Y),!.          (3) edge(a,b).
(2) path(X,Y):- edge(X,Z), path(Z,Y).  (4) edge(a,c).
                                       (5) edge(c,b).
```

Fig. 2.

Example 4. Reconsider a slight variant of the `path` program where a !/0 has been added so that some solutions are prevented (cut away) as depicted in Fig. 2.

If we regard the operational semantics $\mathcal{O}(P, Q)$ as a set of refutations for Q, ordered by the lexicographic order relation $<$, dealing with programs containing a cut resorts to removing from $\mathcal{O}(P, Q)$ those refutations that are "covered" by a cut operation. Informally, we can say that a refutation ρ is covered by a cut operation in a set of SLD-derivations S if:

- there is a derivation $\rho' \in S$ containing a !/0 such that $\rho' < \rho$, and
- ρ and ρ' are identical upto (and including) the parent atom, but differ somewhere between the parent atom and the selection of the !/0 operation in ρ'.

Formalising the above is non-trivial and requires keeping track of the precise relation between each selected atom and its "parent" atom in a single derivation, and between derivations that have a common subderivation. Indeed, dealing with programs containing a cut operation in the operational semantics requires dealing with the SLD-*tree* for a goal Q in P, rather than with the individual refutations for Q. For the example program of Fig. 2, an SLD-tree for the goal $\leftarrow path(a, X)$, constructed using the left-to-right selection rule, is depicted in Fig. 3. The shaded part represents the alternatives that are cut away by the !/0

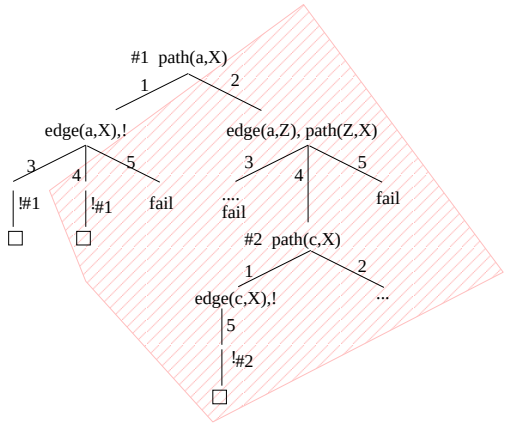

Fig. 3.

in the leftmost branch.

Incorporating this behavior in the fixed point semantics boils down to removing those bottom-up derivations from $\mathcal{F}_A(P)$ that correspond with the answers that are removed when evaluating $\leftarrow A$ by SLD-resolution. In order to trace which bottom-up derivations must be removed from $\mathcal{F}_A(P)$, we first need to incorporate the presence of a !/0 construct into the bottom-up derivations. We treat an occurrence of !/0 in the body of a clause as an atom that is defined by a single clause, namely $! \overset{!}{\leftarrow} \mathit{true}$ that is labelled by the special label '!'. Consequently, occurrences of !/0 in the body of a clause resolve to true and the computed bottom-up derivations contain the term '!' as the label of a (unit) clause. Note that when converting a bottom-up derivation into a characteristic derivation, the relation between an occurrence of !/0 and the label of its parent atom is immediate. Indeed, the cut in an AND-node of the form $l(\tau_1, \ldots, \tau_{k-1}, !, \tau_{k+1}, \ldots, \tau_n)$ refers to the choicepoint that is created by unifying the parent atom with the clause labelled l. We will see further how this observation enables an elegant formulation of the notion of coveredness that was introduced informally earlier.

Example 5. Let P denote the program in Figure 2. The bottom-up derivations constructed by T_P^d are similar to those of Example 3, except for the presence of cuts in the AND-trees:

$$\left\{ \begin{array}{ll} (path(a,b), 1(3,!)) & (path(a,c), 1(4,!)) \\ (path(c,b), 1(5,!)) & (path(a,b), 2(4, 1(5,!))) \end{array} \right\}$$

There is a catch, though. Neither the operational semantics, nor the fixed point semantics introduced before deal with finitely failing derivations. Still, failing derivations must be considered when dealing with programs containing !/0. Consider the following example:

Example 6. Let P be the following program

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

Note that $\mathrm{lfp}(T_P^d) \ni \{(r(b), 1(3, !, 4)\}$. However, $\mathcal{O}(P, \leftarrow r(X)) = \langle\rangle$ since $\leftarrow r(X)$ has a lexicographically smaller derivation that finitely fails but that contains a cut operation that removes the succeeding derivation with answer X/b.

In order to deal with finitely failing derivations in a satisfactory way, it suffices to make failure explicit in the program under consideration. Therefore, instead of computing $\mathrm{lfp}(T_P^d)$, we compute $\mathrm{lfp}(T_{P'}^d)$, where P' is the program that is obtained from P by adding an extra, last clause to every predicate that always succeeds and is labelled by the special label 'fail'.

Example 7. Let P be the program from Example 6, then P' is the following program:

```
(1) r(X):- p(X), !, q(X).        (fail) r(X).
(2) p(a).                        (fail) p(X).
(3) p(b).                        (fail) q(X).
(4) q(b).
```

Note that the introduction of the extra clauses does not alter the successful derivations of the program, where a successful derivation is a derivation that does not contain the label 'fail'[2]. Also note that the introduction of the two special labels – 'fail' and '!'– does not alter the definition of T_P^d, nor the definitions of a bottom-up derivation and its depth-first, left to right traversal. Their presence does make, however, the lexicographic order relation no longer a complete order on the domain of AND-trees (and hence also on $\mathrm{lfp}(T_P^d)$) since the order between '!' or 'fail' and a natural number is undefined. For our purposes, it suffices to consider lexicographic ordering as a partial order, in which $fail > l$ holds for any label $l \neq fail$[3].

Example 8. Let P be the program of Example 7 and consider the goal $\leftarrow r(X)$. We have that

$$\mathcal{F}_{r(X)}(P) = \left\{ \begin{array}{ll} (r(a), 1(2, !, fail)) & (r(b), 1(fail, !, 4)) \\ (r(b), 1(3, !, 4)) & (r(A), 1(fail, !, fail)) \\ (r(b), 1(3, !, fail)) & (r(A), fail) \end{array} \right\}$$

In what follows, we need two important operations on an AND-tree. First, we need to be able to retrieve the i-th label that is encountered when traversing an AND-tree in depth-first, left to right order. Being the i-th element in the depth-first, left to right traversal of the AND-tree (denoted by $\overline{\tau}$), we denote this element by $\overline{\tau}^i$. Next, we need to be able to retrieve the subtree that is found in the i-th node of an AND-tree, when the tree is traversed in a depth-first, from left to right manner. We denote this subtree by $s_i(\tau)$. Formally:

Definition 7 (subtree). *Let τ be an* AND*-tree. We denote with $\#(\tau)$ the number of* AND*-nodes in τ. For any k such that $1 \leq k \leq \#(\tau)$, define s_k recursively as follows:*

$$s_k(l(\tau_1, \ldots, \tau_n)) = \left\{ \begin{array}{ll} l(\tau_1, \ldots, \tau_n) & \text{if } k = 1 \\ s_{k-1-\sum_{j:1}^{i-1} \#(\tau_j)}(\tau_i) & \text{if } k > 1 \end{array} \right.$$

where $i \in \mathbb{N}$ is such that $\sum_{j:1}^{i-1} \#(\tau_j) < k - 1 \leq \sum_{j:1}^{i} \#(\tau_j)$.

Example 9. Consider the AND-tree $\tau = 2(4, 1(5, !))$ taken from Example 5. We have that $s_1(\tau) = \tau$, $s_2(\tau) = 4$, $s_3(\tau) = 1(5, !)$, $s_4(\tau) = 5$, $s_5(\tau) =!$.

It remains to define what bottom-up derivations in $\mathcal{F}_A(P)$ are cut away by a cut contained in some other bottom-up derivation in $\mathcal{F}_A(P)$.

Definition 8 (coveredness). *Let S be a set of* AND*-trees. We say that an* AND*-tree τ is* covered *in S if and only if there exists $\tau' \in S$ and $k > 0$ such that*

[2] Note that the number of failing derivations that effectively needs to be constructed for the semantics can substantially be reduced by pruning during the computation of T_P^d.

[3] The label '!' does not require special treatment: due to the construction of the order relation, a label '!' will only be compared against another label '!'.

- $\forall i, 0 < i < k$, *we have that* $\overline{\tau}^i = \overline{\tau'}^i$, *and*
- $s_k(\tau') = l'(\tau'_1, \ldots, \tau'_n)$ *with* $\tau'_m =!$ *for some* m *in the range* $1, \ldots, n$, *fail does not occur in* $\tau'_1, \ldots, \tau'_{m-1}$ *and either*
  - $s_k(\tau) = l'(\tau_1, \ldots, \tau_n)$ *and* $\exists j : 1 \leq j < m$ *such that* $\tau'_j < \tau_j$ *and* $\tau_1 = \tau'_1, \ldots, \tau_{j-1} = \tau'_{j-1}$, *or*
  - $s_k(\tau) = l(\tau_1, \ldots, \tau_q)$ *with* $l' < l$.

An AND-tree τ is covered in a set of AND-trees if the set contains a lexicographically smaller tree τ' containing a cut such that the depth-first left to right traversals of τ and τ' differ between the cut and the node it refers to. Note that formulating coveredness is easier on the level of the AND-trees than on the level of their depth-first left to right traversals, due to the explicit relation between a cut and its parent node in the notion of an AND-node.

Example 10. Let S denote the set of bottom up derivations depicted in Example 5. We have that $(path(a, c), 1(4, !))$ (denoted by τ_1) is covered in S due to the bottom-up derivations $\tau' = (path(a, b), 1(3, !))$ in S. Indeed, we have $s_1(\tau_1) = 1(4, !)$, $s_1(\tau') = 1(3, !)$ and $3 < 4$ (first case in Definition 8). Likewise, we have that $\tau_2 = (path(a, b), 2(4, 1(5, !)))$ is covered in S due to the same τ'. Indeed, we have that $s_1(\tau_2) = 2(4, 1(5, !)))$, $s_1(\tau') = 1(3, !)$ and $1 < 2$ (second case in Definition 8).

For an atom A, we define $\mathcal{C}_A$ as the subset of $\mathcal{F}_A(P)$ containing the derivations whose associated AND-tree is covered by an AND-tree in $\mathcal{F}_A(P)$. Formally:

Definition 9. *Let* P *be a definite program and* A *an atom. We define*

$$\mathcal{C}_A = \{(B, \tau) \mid (B, \tau) \in \mathcal{F}_A(P) \text{ and } \tau \text{ is covered in } \mathcal{T}\}$$

where $\mathcal{T} = \{\tau \mid (B, \tau) \in \mathcal{F}_A(P)\}$.

At last, we can refine the definition of the bottom-up semantics in order to incorporate programs that contain cut:

Definition 10. *Let* P *be a program and* A *an atom. Then we define*

$$\mathcal{F}_A^{\langle\rangle}(P) = \langle (A_1, \tau_1), (A_2, \tau_2), \ldots \rangle$$

where each $(A_i, \tau_i) \in (\mathcal{F}_A(P) \setminus \mathcal{C}_A(P))$ *and for each* (A_i, τ_i), $(A_j, \tau_j) \in \mathcal{F}_A(P)$: $j > i$ *if and only if* $(A_j, \tau_j) > (A_i, \tau_i)$.

Example 11. Let P denote the program depicted in Figure 2 extended with the clauses labelled 'fail'. We have that

$$\mathcal{F}_{path(X,Y)}(P) = \left\{ \begin{array}{ll} (path(a, b), 1(3, !)) & (path(X, Y), 1(fail, !)) \\ (path(a, c), 1(4, !)) & (path(a, Y), 2(3, fail)) \\ (path(c, b), 1(5, !)) & (path(a, Y), 2(4, fail)) \\ (path(a, b), 2(4, 1(5, !))) & (path(a, Y), 2(3, 1(fail, !))) \\ & \ldots \end{array} \right\}$$

and $\mathcal{C}_{path(X,Y)}$ contains all derivations from $\mathcal{F}_{path(X,Y)}(P)$ but $(path(a,b), 1(3,!))$. Consequently we have that

$$\mathcal{F}^{\langle\rangle}_{path(X,Y)}(P) = \langle (path(a,b), 1(3,!))\rangle.$$

The following example shows the importance of constructing the finitely failing derivations:

Example 12. Let P be the program of Example 7 and consider the goal $\leftarrow r(X)$. $\mathcal{F}_{r(X)}(P)$ is depicted in Example 8. The only non-failing derivation in $\mathcal{F}_{r(X)}(P)$, $(r(b), 1(3,!,4))$, is covered in $\mathcal{F}_{r(X)}(P)$ by the failing derivation $(r(a), 1(2,!,fail))$. Hence, we have that $\mathcal{F}^{\langle\rangle}_{r(X)}(P) = \langle\rangle$, corresponding with $\mathcal{O}(P, \leftarrow r(X))$.

Now, we can extend the result of Theorem 1 to programs that possibly contain cuts as follows.

Theorem 2. *Let P be a definite program that possibly contains cuts and A an atom of the form $p(\overline{X})$. Assume that $\mathcal{F}^{\langle\rangle}_{A}(P) = \langle (A_1, \tau_1), (A_2, \tau_2), \ldots\rangle$ and that $\mathcal{O}(P, \leftarrow A) = \langle \sigma_1, \sigma_2, \ldots\rangle$. Then we have that for all i $A_i \approx A\sigma_i$.*

Again, the theorem states the equivalence of the operational semantics and the fixed point semantics in the presence of cut operations and is extended to regular atoms by the following corollary.

Corollary 3. *Let P be a definite program that possibly contains cuts and let A be an atom. Assume that $\mathcal{F}^{\langle\rangle}_{A}(P) = \langle (A_1, \tau_1), (A_2, \tau_2), \ldots\rangle$ and that $\mathcal{O}(P, \leftarrow A) = \langle \sigma_1, \sigma_2, \ldots\rangle$. Then we have that for all i $A_i\theta_i \approx A\sigma_i$ where $\theta_i = mgu(A, A_i)$.*

5 Discussion

In this work, we have developed a fixed point semantics for pure Prolog programs with cut. We believe that our semantics is simple, elegant and can be the main building block for bottom-up program analyses and transformations that preserve the operational semantics of the transformed program. Our semantics captures the order in which answers are computed, their multiplicity, and deals with pure prolog programs extended with cuts. In the remainder of this section, we first discuss a possible application of our semantics and conclude by discussing related work.

5.1 An application of the semantics

A possible application of the semantics introduced in this paper is the construction of a program transformation that propagates information upwards in a program. Bottom-up propagation of information and the computation of success information has been considered before (e.g. in [19, 15]) and has been advocated as a suitable technique for augmenting top-down partial deduction techniques

(e.g. [17]). In previous work [28, 27] we have developed a complete partial deduction scheme based on an abstraction of the immediate consequence operator. The resulting technique specialises a so-called "open" logic program (i.e. a program that calls a number of predicates that are left undefined) with respect to a definition of the missing predicates.

Such a technique has great potential for speeding up *inductive logic programming* (ILP) techniques in the fields of machine learning and data mining. An ILP problem comprises a background theory, represented by an open logic program, and a database of (positive and negative) examples where each such example is again represented by a logic program that consists of, among others, a particular definition of the predicates that are left undefined in the background knowledge. The task of the ILP system then is to constructed a hypothesis that "explains" the positive examples and rejects the negative ones. The main reason why an ILP application can profit from program specialisation is due to the background theory. Its role is to define a number of additional relations over the problem domain that provide, when combined with a particular example, extra "knowledge" within the context of the example [4]. However, the generality of the background theory can introduce a performance penalty since the search for the best hypothesis involves the evaluation of many queries over each example extended with the background theory. Hence, it makes sense to (partially) specialise the background theory and to use this specialisation for the many query evaluations. A concrete motivating example in the ILP context is the mutagenesis data set [25], where the goal is to learn a rule to predict the mutagenicity[5] of molecules. For this problem, some preliminary experiments that we conducted show that learning from a dataset that was obtained by fully specialising (using hand-crafted techniques) the background knowledge with respect to some examples results in an improvement in speed of a factor 40, while only introducing 15% extra code. Although the ILP setting seems a "traditional" setting suited for program specialisation, it has a number of characteristics that make it hard to apply standard out-of-the-box partial evaluation techniques on it. Unlike the usual approach to partial evaluation, in which the evaluation of a program is restricted to (instances of) a particular query, the first objective of specialising the background knowledge with respect to a particular example is essentially *query-independent*. Rather than propagating data provided in a query downwards in a program – performing as much as the unifications as possible underway – one aims at partially precomputing the relations in the background theory by propagating the information that resides in the predicate definitions that constitute a particular example upwards in the background theory.

A limitation of the bottom-up techniques from [28, 27] is that they do not preserve multiplicity of answers, nor the order in which solutions are computed. In addition, they are not capable of dealing with programs that contain cuts. This is a major disadvantage if these (or other) techniques are to be applied to

[4] In fact, the ability to express background knowledge is one of the main advantages in expressivity that ILP offers over other, less expressive formalisms for data mining.

[5] Mutagenicity is the ability to mutate DNA, which is a possible cause of cancer.

ILP problems that usually rely strongly on these operational characteristics of Prolog. In the remainder of this section, we sketch how the semantics that is presented in this work could be abstracted into a transformation that performs bottom-up partial deduction. A full formal development of the transformation is outside the scope of this paper.

In general, $\mathrm{lfp}(T_P^d)$ will not be finite and one should endorse T_P^d with abstraction in order to compute a finite set of *partial* bottom-up derivations rather than an infinite set of complete bottom-up derivations. In previous work [28, 27] we have shown that this can be achieved by redefining the immediate consequence operator over a domain of clauses, rather than atoms (like e.g. in [5, 6]) and combining it with an abstraction function that replaces a clause $A \leftarrow \overline{B}$ that represents a partial derivation under construction with a tautology $H \leftarrow H$, where H is a generalisation of A. In this work, we take a slightly different approach. First, we extend the notion of a labelled *and*-tree such that its leaf nodes are either labelled AND-nodes with no children, denoted by $l()$ (as before) or atoms, denoted by (H). The idea is that, like a labelled AND-tree in a bottom-up derivation (A, τ) represents the computation that was performed to derive A, an atom in a leaf of τ represents a placeholder for some other computation that is required to compute A. We will use the notation $\mathit{leaves}(\tau)$ to denote the sequence of atoms that is obtained by traversing a labelled AND-tree depth first, from left to right. As such, the bottom-up derivation becomes a partial derivation and the answers computed by (A, τ) become dependent on the answers computed by the atoms in the leaves of τ. If no control information was to be exploited, the partial derivation could as well be represented by a clause $A \leftarrow \mathit{leaves}(\tau)$. The reason for keeping the atoms in τ rather than directly deriving a clause is in particular due to the handling of cuts, which we treat later on.

In what follows, we will denote the combination of an atom with such an extended AND-tree as a *clausal bottom-up derivation* and use $\mathcal{CBU}$ to denote the set of all such clausal bottom-up derivations. Likewise, we introduce $T_P^{cl} : \wp(\mathcal{CBU}) \mapsto \wp(\mathcal{CBU})$, whose definition equals the definition of T_P^d; the only difference being the domain $\mathcal{CBU}$ rather than $\mathcal{BU}$.

To convert the computation of a possibly infinite set of bottom-up derivations into the computation of a *finite* set of *partial* bottom-up derivations, we introduce the notion of an abstraction function. Such an abstraction function maps a set of clausal bottom-up derivations onto a set of clausal bottom-up derivations in the following way:

Definition 11. *An idempotent function $f : \wp(\mathcal{CBU}) \mapsto \wp(\mathcal{CBU})$ is an* abstraction function *if $\forall S \in \wp(\mathcal{CBU})$ it holds that if $(H, \tau) \in S$, then either $(H, \tau) \in f(S)$ or $\exists (H', (H')) \in f(S)$ such that $H' \preceq H$.*

Note that in Definition 11, $(H', (H'))$ represents a special bottom-up derivation consisting of an atom H' with an associated AND-tree that consists of a single (leaf) node labelled by the atom itself. The central idea behind abstracting a set of (clausal) bottom-up derivations is that a derivation (H, τ) can be dropped from the set if there exists a derivation $(H', (H'))$ in the set with H' a more general atom than H. When abstraction is combined with T_P^{cl}, this reflects the

fact that the derivation (H, τ) is no longer to be extended from H, but a new bottom-up derivation is started from a more general atom H'.

The computation of $\mathrm{lfp}(T_P^d)$ can then be abstracted by the computation of a finite sequence of sets of clausal bottom-up derivations $S_0, S_1, \ldots, S_n$ such that $S_0 = \emptyset$ and for all i,

$$S_i = f(T_P^{cl}(S_{i-1}))$$

and $S_n = S_{n-1}$ with f being an abstraction function. It can be proven – under the appropriate conditions [28, 27] – that, when the clauses from S_n are recombined with the clauses that were dropped during the subsequent applications of f in the computation of the sequence $S_0, S_1, \ldots, S_n$, one obtains a program P' for which the *declarative* semantics equals the declarative semantics of the original program, that is $\mathrm{lfp}(T_{P'}) = \mathrm{lfp}(T_P)$. If that is the case, we say that S_n is a *suitable* abstraction of $\mathcal{F}(P)$. In general, we will use $\mathcal{F}^{cl}(P)$ to denote a suitable abstraction of $\mathcal{F}(P)$.

The question that remains is how to convert the clausal bottom-up derivations in $\mathcal{F}^{cl}(P)$ into a residual program P' such that for a particular goal Q, $\mathcal{O}(P, Q) = \mathcal{O}(P', Q)$. First, note that the partial order relation defined over bottom-up derivations remains defined over the domain of clausal bottom-up derivations. Hence, for an atom A, we can define $\mathcal{F}_A^{cl}(P)$ in a similar way as $\mathcal{F}_A^{\langle\rangle}(P)$. Note however that, unlike $\mathcal{F}_A^{\langle\rangle}(P)$, $\mathcal{F}_A^{cl}(P)$ is not unique, in the sense that it depends on a particular abstraction function. Definition 6 can be refined into:

Definition 12. *Let P be a program, A an atom of the form $\leftarrow p(\overline{X})$ and $\mathcal{F}^{cl}(P)$ a suitable abstraction of $\mathcal{F}(P)$. We define*

$$\mathcal{F}_A^{\langle\rangle}(P) = \langle (A_1, \tau_1), \ldots, (A_n, \tau_n) \rangle$$

where each $(A_i, \tau_i) \in \mathcal{F}^{cl}(P)$ and for each $(A_i, \tau_i), (A_j, \tau_j) \in \mathcal{F}^{cl}(P) : j > i$ if and only if $(A_j, \tau_j) > (A_i, \tau_i)$.

Now, if no abstraction was employed during the computation of $\mathcal{F}^{cl}(P)$, none of the leaves of the bottom-up derivations are atoms and each bottom-up derivation $(A', \tau) \in \mathcal{F}_A^{\langle\rangle}(P)$ effectively constitutes an answer for A. Consequently, one can generate a specialised program for A by removing those derivations that are covered by a cut as before and generating a unit clause for each remaining bottom-up derivation. If abstraction was employed, on the other hand, some of the bottom-up derivations contain atoms as their leaves, effectively representing a placeholder for a computation and one should generate plain clauses rather than unit clauses in the residual program. This, however, requires a different treatment of the cuts that are possibly present in the bottom-up derivations. Indeed, the presence of an atom in a bottom-up derivation can make the effect of a cut undecidable at transformation-time since the atom represents a placeholder for (a number of) computations of which the effect is unknown. Consequently, if the atom is in the scope of a cut, both the computation represented by the atom and the cut must be residualised in the transformed program such that the cut

exhibits the desired behavior – depending on the residualised computations – in the transformed program.

5.2 Related work

Operational and fixed point semantics of logic programs is a well-studied research field. In [4], an operational semantics is presented that models the meaning of a goal by a sequence of computed answers. The semantics handles programs with cut and contains a bottom element that represents infinite looping. The semantics characterises a program by functional equations and is mainly developed towards proving termination of logic programs. Also [22] defines an operational semantics by means of a sequence of answers and uses it to show that a number of transformation rules (unfold/fold) preserve this semantics. It does not deal with programs containing cuts. In [2], Prolog control is explicitly modeled in a constraint logic language and a declarative and operational semantics is presented. It encodes the Prolog search rule, but does not deal with programs containing cuts.

In [10], a semantics expressed in term of SLD-derivations is defined. The semantics of a program is goal-independent and various properties of SLD-derivations are studied. The main focus is on compositionality of the semantics, which does not deal with programs containing cuts. A semantics that is perhaps more related to our work is [11]. It defines a denotational semantics for Prolog programs that contain cuts. The meaning of a program is defined by a number of semantic functions on the syntactic constructs of a program that can be used to compute a goal-independent denotation of the program. The work is motivated mainly by the need to verify a number of transformations. The main difference with our work, is that we obtain a denotation of the program by applying a bottom-up immediate consequences operator, which seems to be more appropriate in our context since the main motivation of our work lies in abstracting the semantics to obtain a bottom-up partial evaluation scheme. Also closely related is [24]. It defines a general framework for goal-independent abstract semantics of Prolog that also models the depth-first search rule and the cut. The semantics consists of a sequence of pairs, of which one part denotes a computed answers while the other part – the so called "observability constraints" – give information about the cut executions. One of the main distinctions between our approach and the one of [24] is that in the latter work, the denotation consists of AND-OR trees, rather than AND-trees alone. Although this makes the technique interesting and applicable in a broad number of situations, the semantics also relies on (substantially complicated) tree operations to combine elements from the computed denotation. We believe that for our particular application – the creation of a 2-phase partial evaluator for Prolog – a semantics that is particularly targeted towards this goal might be preferable.

In [8], the authors present a framework for the abstract interpretation of Prolog that handles the depth-first search rule and cut. It models program execution by sequences of substitutions that represent the sequences of computed answer substitutions returned by a goal $\leftarrow p(\overline{X})$. Contrary to our approach, however,

the framework is essentially an adaptation of top-down frameworks for abstract interpretation. Finally, the framework of [13] defines an operational abstract semantics. It decorates abstract OLDT trees with extra control information, that is later on used by the !/0 operations to prune the OLDT tree.

Acknowledgements

We thank Mike Codish, Alberto Pettorossi and Marco Comini for stimulating discussions, and anonymous referees for their extensive and constructive comments on a draft of the current paper.

References

1. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers B.V., 1990.
2. Roberto Barbuti, Michael Codish, Roberto Giacobazzi, and Giorgio Levi. Modelling Prolog control. *Journal of Logic and Computation*, 3(6):579–603, December 1993.
3. Roberto Barbuti, Roberto Giacobazzi, and Giorgio Levi. A General Framework for Semantics-Based Bottom-Up Abstract Interpretation of Logic Programs. *ACM TOPLAS*, 15(1):133–181, January 1993.
4. Marianne Baudinet. Proving termination of Prolog programs: A semantic approach. *Journal of Logic Programming*, 14(1 & 2):1–29, 1992.
5. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19/20:149–197, 1994.
6. A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.
7. M. Bugliesi and F. Russo. Partial evaluation in Prolog: Some improvements about cut. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 645–660, Cleveland, Ohio, USA, 1989.
8. Baudouin Le Charlier, Sabina Rossi, and Pascal Van Hentenryck. Sequence-based abstract interpretation of Prolog. *Theory and Practice of Logic Programming*, 2(1):25–84, 2002.
9. M. Codish, D. Dams, and E. Yardeni. Bottom–Up Abstract Interpretation of Logic Programs. *Theoretical Computer Science*, 124(1):93–125, February 1994.
10. Marco Comini and Maria Chiara Meo. Compositionality properties of SLD-derivations. *Theoretical Computer Science*, 211(1 & 2):275–309, January 1999.
11. S. K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, 1988.
12. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic programs. *Theoretical Computer Science*, 69:289–318, 1989.
13. G. Filè and S. Rossi. Static analysis of Prolog with cut. In A. Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programmng and Automated Reasoning*, pages 134–145, 1993. Springer-Verlag.

14. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
15. M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming H Languages, Implementations, Logics and Programs (PLILP'96)*, pages 137–151, Aachen, Germany, 1996. Springer-Verlag. LNCS 1140.
16. Michael Leuschel. Partial evaluation of the "real thing". In Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'94*, Pisa, Italy, 1994.
17. Michael Leuschel. Program specialisation and abstract interpretation reconciled. In Joxan Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 220 – 234, Manchester, UK, 1998. MIT Press.
18. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
19. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 909–923, Seattle, 1988. ALP, IEEE, The MIT Press.
20. R. A. O'Keefe. On the treatment of cuts in Prolog source-level tools. In *Proceedings of the International Symposium on Logic Programming*, pages 68–72. IEEE Computer Society, Technical Committee on Computer Languages, The Computer Society Press, July 1985.
21. S. Prestwich. An unfold rule for full Prolog. In Kung-Kiu Lau and Tim Clement, editors, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation*, Workshops in Computing, pages 199–213, London, July 2–3 1993. Springer Verlag.
22. M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *Proceedings of PEPM'91*, Sigplan Notices, Vol. 26, N. 9, pages 274–284, 1991.
23. D. Sahlin. Mixtus: An automatic partial evaluator for full prolog. *New Generation Computing*, 12(1):7–51, 1993.
24. F. Spoto. Operational and goal-independent denotational semantics for Prolog with cut. *Journal of Logic Programming*, 42(1):1–46, 2000.
25. Ashwin Srinivasan, Stephen Muggleton, Michael J. E. Sternberg, and Ross D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1-2):277–299, 1996.
26. M. H. van Emden and R. Kowalski. The Semantics of Predicate Logic as Programming Language. *Journal of ACM*, 23(4):733–743, 1976.
27. W. Vanhoof, D. De Schreye, and B. Martens. A framework for bottom up specialisation of logic programs. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the Joint International Symposia PLILP/ALP 1998*, volume 1490 of *Lecture Notes In Computer Science*, pages 54–72. Springer-Verlag, 1998.
28. W. Vanhoof, D. De Schreye, and B. Martens. Bottom-up partial deduction of logic programs. *The Journal of Functional and Logic Programming*, 1999:1–33, 1999.