Proving completeness of logic programs with the cut

Włodzimierz Drabent¹,²

¹ Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
² Linköping University, Linköping, Sweden

Abstract. Completeness of a logic program means that the program produces all the answers required by its specification. The cut is an important construct of programming language Prolog. It prunes part of the search space, this may result in a loss of completeness. This paper proposes a way of proving completeness of programs with the cut. The semantics of the cut is formalized by describing how SLD-trees are pruned. A sufficient condition for completeness is presented, proved sound, and illustrated by examples.

Keywords: Logic programming, The cut, Operational semantics, Program completeness, Program correctness

1. Introduction

Some constructs of programming language Prolog prune part of the search space, i.e. of an SLD-tree under the Prolog selection rule (LD-tree). The basic pruning construct is the cut. Pruning does not change the declarative meaning of a program; the program treated as a set of logic formulae is the same with and without pruning constructs. What is changed is the operational semantics—the way the program is executed. As pruning means skipping some fragments of the search space, it may result in Prolog missing some answers. This paper presents a way of proving completeness of programs with the cut, i.e. proving that a Prolog program would produce all the answers required by its specification.

The work has to be based on a formal semantics. Usually the semantics of the cut is described in terms of explicit representation of computation states, stacks of backtrack points, numerical labels related to cut invocations etc, like in [Bil90, dV89, SGS+10, And03]. Some approaches require transforming programs into a special syntax [Bil90, KK14], or restrict the class of programs dealt with ([KK14] requires so-called cut-stratification). Some approaches describe only approximations of the semantics. The semantics of [SGS+10] does not distinguish success from failure, as the purpose of the semantics is termination analysis. The semantics of [KK14] may describe answers which actually are not computed; such inaccuracy is acceptable as the semantics is intended as a basis for abstract interpretation, which introduces inaccuracies anyway. Of course such semantics is inadequate for reasoning about program completeness.

Correspondence and offprint requests to: W. Drabent, E-mail: drabent@ipipan.waw.pl
In this paper we define the semantics of programs with the cut in terms of pruning LD-trees. Such approach is convenient—the main proof of this paper is based on comparing pruned and non-pruned LD-trees. It is also closer to the usual way of describing operational semantics (in terms of SLD-resolution) than the approaches mentioned above. Two approaches somewhat similar to ours are those of Apt [Apt97] and Spoto [Spo00]. They also employ trees, but the trees are not defined as subgraphs of the LD-tree. In [Apt97], initial queries containing the cut seem not considered. The approach of [Spo00] seems more complicated than ours and that of [Apt97]. For instance, new cut symbols are added, to embed in tree nodes information about the origin of each cut. Our formal semantics considers definite clause programs with the cut, and Prolog selection rule. Similarly to [Apt97, Spo00], it does not deal with modifying the selection rule by means of delays (also called coroutining). Other control constructs, like the conditional or negation as failure, can be expressed by means of the cut.

Little work has been done on reasoning about completeness of logic programs, see [Dra16a] (or [Dra15]), [DM93], and the references therein; see also Sect. 4.1 of this paper for a particular completeness proving method.

An approach to proving completeness in presence of pruning is presented in [Dra16a] (also reported in [Dra15]). It is based on a more abstract view of pruning than this work. It does not directly refer to a pruning construct in programs, like the cut. For a completeness proof, one has to separately find out which clauses are applied to each selected atom in a pruned SLD-tree. Determining the clauses may be not obvious, as a single invocation of the cut may result in pruning children of many nodes of a tree. Moreover, different numbers of children may be pruned for nodes with similar selected atoms. On the other hand, the approach is not restricted to the selection rule of Prolog, and applies to any kind of pruning. The author is not aware of any other work on proving properties of programs with the cut, particularly on proving completeness.

A related subject is abstract interpretation; for its applications to programs with the cut see [SGS+10, KK14] and the references therein. In abstract interpretation, properties of programs are derived automatically, however the class of possible properties is restricted to the chosen abstract domain.

The main result of this paper is a sufficient condition for completeness of LD-trees pruned due to the cut. (As completeness depends on initial queries we formally do not talk about program completeness, but completeness of trees.) The sufficient condition is proved sound w.r.t. the formal semantics. It is illustrated by a few examples. A preliminary version of the sufficient condition, restricted to the cut in the last clause of a procedure, appeared in [Dra15].

Pruning constructs, like the cut, may destroy completeness of programs, but they preserve program correctness. However it is possible that a logic program is incorrect, but behaves correctly (for some initial queries) under pruning, as wrong answers are pruned. Such programming technique is called “red cut”. Proving correctness in such case is outside of the scope of this paper, and is a subject of future work. See [Dra16a] for a sufficient condition for correctness in a context of the other approach to pruning. Another subject of future work is dealing with other selection rules (Prolog with delays).

Let us outline the rest of the paper. The next section is an overview of basic concepts. Section 3 deals with the operational semantics, first discussing LD-resolution and then introducing LD-resolution with the cut. Section 4 discusses proving correctness and completeness of programs without pruning. In particular, it discusses a specific notion of correctness related to the operational semantics (LD-resolution); this notion is needed in the next section. Section 5 presents a sufficient condition for completeness in the presence of the cut. Section 6 presents some example proofs of program completeness. The Appendix contains proofs missing in Sect. 5.

2. Preliminaries

In this paper we consider definite clause programs (informally—logic programs without negation), and Prolog programs that are definite clause programs with (possibly) the cut. We assume that the reader is familiar with basics of Prolog, and basics of the theory of logic programming, including the notions of (Herbrand) interpretation/model, logical consequence, (definite clause) program, query, the least Herbrand model, substitution, most general unifier, SLD-derivation, SLD-tree, and soundness/completeness of SLD-resolution [NM95, Llo87, Apt97]. We follow the definitions and notation of [Apt97], unless stated otherwise. In particular, the elements of SLD-derivations and nodes of SLD-trees are queries, i.e. conjunctions of atoms, represented as sequences of atoms. (Instead of queries, the other approach [NM95, Llo87] uses goals, i.e. negations of queries.) LD-resolution (LD-derivation, LD-tree) is SLD-resolution (SLD-derivation, SLD-tree) with Prolog selection rule—in any query its first atom is selected; see also Sect. 3.1.
Following [Apt97], we assume that truth of a formula is defined in such a way that \( I \models F \) iff \( I \models \forall F \) (for any formula \( F \) and any interpretation \( I \)). An atom whose predicate symbol is \( p \) will be called a \( p \)-atom, or an atom for \( p \). Similarly, a clause whose head is a \( p \)-atom is a clause for \( p \). In a program \( P \), by procedure \( p \) we mean the set of clauses for \( p \) in \( P \).

We do not require that the considered alphabet consists only of the function and predicate symbols occurring in the considered program. The Herbrand universe (i.e. the set of ground terms) will be denoted by \( \mathcal{H}U \), the Herbrand base (the set of ground atoms) by \( \mathcal{HB} \), and the sets of all terms, respectively, atoms, by \( \mathcal{TB} \) and \( \mathcal{TB} \). The least Herbrand model of a program \( P \) will be denoted by \( \mathcal{M}P \). For an expression \( E \) (i.e. a term, a formula, or a sequence of terms or formulae), by \( \text{ground}(E) \) we mean the set of ground instances of \( E \). For a program \( P \), \( \text{ground}(P) \) is the set of ground instances of the clauses of \( P \). The set of variables occurring in an expression \( E \) will be denoted \( \text{vars}(E) \). Notation \( \text{vars}(E) \), \( \text{ground}(E) \) will not be used for expressions containing quantifiers.

By an answer for a program \( P \) we mean a query \( Q \) such that \( P \models Q \). (In [Apt97] answers are called “correct instances of queries”.) By a computed answer for a program \( P \) and a query \( Q \) we mean an instance \( Q \theta \) of \( Q \) where \( \theta \) is a computed answer substitution [Apt97] obtained from some successful SLD-derivation for \( Q \theta \) and \( P \). Often it is not necessary to distinguish answers and computed answers, as by soundness and completeness of SLD-resolution, for any given selection rule, \( Q \) is an answer for \( P \) iff \( Q \) is a computed answer for \( P \) (and some query).

Names of variables begin with an upper-case letter. For a substitution \( \theta = \{X_1/t_1, \ldots, X_n/t_n\} \), we denote \( \text{dom}(\theta) = \{X_1, \ldots, X_n\} \), \( \text{rng}(\theta) = \text{vars}(t_1, \ldots, t_n) \), and \( \text{vars}(\theta) = \text{dom}(\theta) \cup \text{rng}(\theta) \). The substitution \( \theta \) is ground if \( t_1, \ldots, t_n \) are ground terms. Note that if \( \theta, \sigma \) are ground substitutions with disjoint domains, that is \( \text{dom}(\theta) \cap \text{dom}(\sigma) = \emptyset \), then \( \theta \sigma = \theta \cup \sigma = \sigma \theta \). The restriction of \( \theta \) to a set \( V \) of variables is \( \theta|_V = \{X/t \in \theta \mid X \in V\} \). For an expression \( E \), by \( \theta|_E \) we mean \( \theta|_{\text{vars}(E)} \). The empty substitution is denoted by \( \epsilon \). A substitution \( \theta \) is idempotent when \( \text{dom}(\theta) \cap \text{rng}(\theta) = \emptyset \). Abbreviation mgu stands for “most general unifier”. A unifier \( \theta \) of expressions \( E_1 \), \( E_2 \) is relevant if \( \text{vars}(\theta) \subseteq \text{vars}(E_1) \cup \text{vars}(E_2) \).

We use the list notation of Prolog. So \([t_1, \ldots, t_n] \ (n \geq 0) \) stands for the list of elements \( t_1, \ldots, t_n \). Only a term of this form is considered a list. (Thus terms like \([a, a \mid X] \), or \([a, a \mid a] \) where \( a \) is distinct from \([] \), are not lists). Sometimes, in examples, we will use the Prolog symbol \( [\cdot] \) instead of \( \leftarrow \) in programs. The set of natural numbers will be denoted by \( \mathbb{N} \).

3. Semantics for definite clause programs with the cut

This section formalizes a main part of the semantics of Prolog. We present an operational semantics of definite clause programs augmented with the cut (!). First we abstract from the cut, describing LD-resolution. Then we describe how the cuts prune LD-trees. We begin with a note of declarative semantics.

To incorporate the cut into programs, let us add a new 0-argument predicate symbol \( ! \) to the alphabet, and extend the set \( \mathcal{TB} \) of atoms: \( \mathcal{TB}^+ = \mathcal{TB} \cup \{!\} \). A program with cuts is a finite sequence of definite clauses of the form \( H \leftarrow B_1, \ldots, B_n \), where \( n \geq 0 \), \( H \in \mathcal{TB} \), and \( B_1, \ldots, B_n \in \mathcal{TB}^+ \).

In the rest of the paper we write “program” for “program with cuts”. Sequences of atoms from \( \mathcal{TB}^+ \) will often be denoted by \( A, B \) etc., with possible indices. We do not introduce any notation to refer to the elements of such sequences, so e.g. \( A \) (or \( A_i \)) is unrelated to \( A \). When this does not lead to ambiguity, we sometimes treat queries as sets of atoms, and programs with cuts as sets of clauses, and e.g. write that a clause is a member of a program, or write \( A \subseteq S \) to say that each atom of the sequence \( A \) is in the set \( S \).

**Declarative semantics** When considering programs from the point of view of logic, atom \( ! \) will be treated as true in each interpretation. Thus \( I \models A, ! \) iff \( I \models A, \bar{B} \) (where \( \bar{A}, \bar{B} \subseteq \mathcal{TB}^+ \)). So, in what follows we assume that interpretations do not describe the semantics of \( ! \). Hence by an Herbrand interpretation we mean a set of ground atoms from \( \mathcal{HB} \). Assume that a definite program \( P' \) is a program with cuts \( P \) with each \( ! \) removed. Then \( P, P' \) have the same models, the same Herbrand models, and thus the same least Herbrand model and the same answers.

---

1 Formally, \([t_1, \ldots, t_n]\) is an alternative notation for the term \( \bullet(t_1, \bullet(t_2, \ldots, \bullet(t_n, []))\ldots) \), where \( \bullet \) is the list constructor and \([] \) is the empty list. Also, \( [t \mid u], [s, t \mid u] \) is an alternative notation for, respectively, \( \bullet(t, u), \bullet(s, \bullet(t, u)) \).
3.1. LD-resolution

For our purposes we need a slight generalization of the standard LD-resolution for programs with the cut. The role of the cut is pruning LD-trees. So we first consider LD-resolution, where the cut is neglected, and then we introduce the semantics of the cut by defining how LD-trees are pruned.

An LD-derivation for a program \( P \) is a pair of (finite or infinite) sequences: a sequence \( Q_0, Q_1, \ldots \) of queries, and a sequence \( \theta_1, \theta_2, \ldots \) of mgus. (The sequences are either both infinite, or both finite with respectively \( n + 1 \) and \( n \) elements, \( n \geq 0 \).) If \( Q_{i-1} = \lambda \), then \( Q_i = \lambda \) and \( \theta_i = \epsilon \) (the empty substitution). Otherwise the successor of \( Q_{i-1} \), if any, is as in the standard LD-resolution: If \( Q_{i-1} = Q \), then \( Q_i = (\lambda, \lambda) \theta_i \), where \( \theta_i \) is an mgu of \( A \) and \( H \) and \( H \leftarrow \lambda \) is a standardized apart variant\(^2\) of a clause \( C \) of the given program. Without loss of generality we can assume that the employed mgu’s are idempotent and relevant. An LD-derivation \( Q_0, \ldots, Q_n ; \theta_1, \ldots, \theta_n \) is successful if its last query is empty. The (computed) answer of such derivation is \( Q_0 \theta_1 \cdots \theta_n \).

As a query \( Q \) may occur in a derivation \( D \) a few times, one should speak about occurrences of queries in derivations. The same for an atom in a query, an atom selected in a derivation, etc. However, to simplify the presentation, we usually skip the word “occurrence”.

The notion of derivation described above is slightly different from those of [Apt97] and [Doe94]. In [Apt97] a proper prefix of a derivation is not a derivation, while here it is. In [Doe94] the substitutions of a derivation are not the mgu’s, but specializations. (Roughly speaking, specializations are mgu’s restricted to the variables of queries; instead of \( \theta_i \) there is a specialization \( \theta_i |_{Q_{i-1}} \).

Consider an LD-derivation \( D \) containing a query \( Q_j = \lambda \). Following [DM88], we describe a fragment of \( D \) which may be viewed as the evaluation of \( \lambda \).

**Definition 3.1** (subderivation) Let \( D = Q_0, Q_1, \ldots; \theta_1, \theta_2, \ldots \) be a (finite or infinite) LD-derivation, and \( Q_i = \lambda \), \( \lambda \) a query in \( D \). If \( D \) contains a query \( Q_m = \lambda \theta_{j+1} \cdots \theta_m \), where \( m > j \), then \( \lambda \) (of \( Q_j \)) succeeds in \( D \).

If \( \lambda \) (of \( Q_j \)) does not succeed in \( D \) then the subderivation of \( D \) for \( \lambda \) (of \( Q_j \)) is the (finite or infinite) derivation \( D_j = Q_j, Q_{j+1}, \ldots, \theta_{j+1}, \theta_{j+2}, \ldots \) that contains each query \( Q_i \) and substitution \( \theta_{i+1} \) of \( D \) such that \( i \geq j \).

If \( \lambda \) (of \( Q_j \)) succeeds in \( D \) then the subderivation of \( D \) for \( \lambda \) (of \( Q_j \)) is the derivation \( D_j = Q_j, \ldots, Q_m; \theta_{j+1}, \ldots, \theta_m \), where \( Q_m = \lambda \theta_{j+1} \cdots \theta_m \) and, for \( i = j, \ldots, m-1 \), \( Q_i = \lambda \theta_{j+1} \cdots \theta_i \) with nonempty \( \lambda \). Such subderivation is called successful, and \( \lambda \theta_{j+1} \cdots \theta_m \) is called the (computed in \( D \)) answer for \( \lambda \).

A subderivation for an atom \( p(\bar{t}) \) of a query \( p(\bar{t}) \), \( \lambda \) within a derivation \( D \) may be informally understood as a procedure invocation (of procedure \( p \)). In an extreme case of empty \( \lambda \) (i.e. \( Q_j = \lambda \)), the subderivation for \( \lambda \) of \( Q_j \) consists of a single query \( Q_j \) (and no substitutions). Note that, given \( D \) and \( Q_j = \lambda \), \( \lambda \), a subderivation for \( \lambda \) is unique. Due to the clauses being standardized apart and the mgu’s being relevant, we have:

**Lemma 3.2** Let \( D \), \( Q_j \) and \( D_j \) be as in the definition above \( (D \) an LD-derivation, \( Q_j = \lambda \), \( \lambda \) a query of \( D \), and \( D_j \) be the subderivation for \( \lambda \) starting at \( Q_j \)). Assume that a variable \( X \) occurs in \( Q_0, \ldots, Q_j; \theta_1, \ldots, \theta_j \) or in a clause variant used to derive some of \( Q_1, \ldots, Q_j \), and that \( X \) does not occur in \( \lambda \). Then \( X \) does not occur in any \( \theta_{j+1}, \theta_{j+2}, \ldots \) of \( D_j \). Neither it occurs in the prefix \( \lambda \), of any query \( Q_i = \lambda \theta_{j+1} \cdots \theta_i \) of \( D_j \) (for \( i > j \)).

**Proof** By induction on \( i \) (as the clauses employed in \( D \) are standardized apart, and the mgu’s are relevant).

The LD-tree for a program \( P \) and a query \( Q \) is defined in a standard way. The root of the tree is \( Q \) and its branches are LD-derivations. A node \( Q' \) to which \( k \) clauses of \( P \) are applicable, has \( k \) children, one for each such clause of \( P \). The ordering of the children follows that of the clauses in \( P \). See also Example 3.4 below. Formally, LD-trees are trees with nodes labelled with queries. We will often simply say that a node is a query, taking care that this does not lead to ambiguities.

---

\(^2\) This means that no variable of \( H \leftarrow \lambda \) occurs in \( Q_0, \ldots, Q_{i-1}, \theta_1, \ldots, \theta_{i-1} \), or in a clause variant used in deriving some \( Q_j \), for \( 0 < j < i \). If (some variant of) the head of \( C \) is unifiable with \( A \) then we say that \( C \) is applicable to query \( A, \lambda \).
3.2. Semantics of the cut

LD-trees are the search spaces of Prolog computations. The role of the cut is to skip the search of some subtrees of an LD-tree. We first formalize the order in which Prolog searches LD-trees. Note that the part of the tree to the right of (any) infinite path is not searched.

Definition 3.3 The preorder sequence seq(T) of nodes of (an ordered) tree T is defined recursively as

\[ seq(T) = Q, seq(T_1), \ldots, seq(T_n) \]

where Q is the root of T, with the children \( Q_1, \ldots, Q_n \) (in this order), \( 0 < i \leq n \) or \( 0 = i = n \).

\( T_1, \ldots, T_n \) are the subtrees of T rooted in \( Q_1, \ldots, Q_n \), respectively.

\( T_1, \ldots, T_{i-1} \) are finite.

\( i = n \) or \( T_i \) is infinite.

Before introducing the formal semantics of the cut, we describe the semantics informally and illustrate it by an example. Consider an LD-tree T and its node \( Q'' \) with the cut selected (i.e. ! is the first atom of \( Q'' \)). When Prolog visits \( Q'' \), some nodes of T are pruned; in other words, they will not be visited in the further search of T. The pruned nodes are (some) descendants of \( Q'' \)—the node which introduced the cut of \( Q'' \). (The cut appeared first in a child of \( Q'' \).) All the descendants to the right of the path \( Q', \ldots, Q'' \) are pruned.

Example 3.4 Consider the program and the LD-tree from the diagram. The cut introduced by node q, ! and executed in node !, r, ! prunes the descendants of the nodes of the path (q, !), (s, !, r, !), (!, r, !) to the right of the path. (The descendants of the last node !, r, ! are not to the right of the path, and are not pruned.) So after visiting the node !, r, ! Prolog visits nodes r, ! and r.

The pruning can also be viewed as follows. Assume a clause \( C \) is applied to a node \( Q_i \) of the path to obtain the next node of the path. Then no clause following \( C \) in the program is applied to \( Q_i \).

Assume now that the rule \( q : - s, !, r \) is removed from the program. Hence node s, !, r, ! and its descendants are removed from the tree. Now the cut in node ! is executed. This prunes the nodes to the right of the path (p), (q, !), (t, !), (!), namely r, r, ! and r. 

The next two definitions describe the semantics of the cut.

Definition 3.5 Consider an LD-tree T with a branch D containing consecutive nodes

\[ Q_{j-1} = A, \tilde{A} \]

\[ Q_j = (\tilde{B}_1, !, \tilde{B}_2, \tilde{A})\theta_j \]

\[ \ldots \]

\[ Q_m = (!, \tilde{B}_2, \tilde{A})\theta_j \cdots \theta_m \]

such that \( j \leq m \), and \( Q_{j-1}, \ldots, Q_m; \theta_j, \ldots, \theta_m \) is a subderivation of D for \( \tilde{B}_1 \).\(^3\) We say that the node \( Q_{j-1} \) introduces the cut of \( Q_j \), and that the cut of \( Q_j \) is potentially executed in the node \( Q_m \).\(^4\) Derivation \( Q_{j-1}, \ldots, Q_m \) is called a cutting sequence of nodes in T. Its first node \( Q_{j-1} \) will be called the introducing node, and its last node \( Q_m \)—the executing node of the cutting sequence.

\(^3\) Thus a clause variant \( H \leftarrow \tilde{B}_1, !, \tilde{B}_2 \) was applied to \( Q_{j-1} \), and, for \( i = j, \ldots, m - 1 \), each \( Q_i \) is of the form \((\tilde{A}_i, !, \tilde{B}_2, \tilde{A})\theta_j \cdots \theta_i \) with nonempty \( \tilde{A}_i \).

\(^4\) We write “the cut of \( Q_j \)” for brevity. Formally we deal here with the occurrence of ! between \( \tilde{B}_1 \theta_j \) and \( \tilde{B}_2 \theta_j \) in the node \( Q_j \). When there are more such occurrences, the objects introduced by this definition are defined separately for each occurrence of ! in \( Q_j \).
For a case where the cut occurs in the initial query \((j = 0, \theta_0 = \epsilon, \text{ and } Q_{j-1} \text{ does not exist})\), “potentially executed” is defined as above, and the cutting sequence is \(Q_0, \ldots, Q_m\). (Note that such cutting sequence does not have its introducing node.)

Each node of the form \(!, \bar{A}\) in the tree is the executing node of a unique cutting sequence; the sequence will be called the cutting sequence of the executing node \(!, \bar{A}\).

The nodes of \(T\) pruned by a cutting sequence \(Q_{j-1}, \ldots, Q_m\) are those children of each \(Q_i\) that are to the right of \(Q_i\), for \(i = j, \ldots, m\), and the descendants of the children. The nodes pruned by an executing node \(Q_m\) are the nodes pruned by the cutting sequence of \(Q_m\).

**Example 3.6** In the tree from Example 3.4, the executing nodes are !, r, ! and !. The cutting sequence of !, r, ! is \((q, !), (s, !, r, !), (!, r, !)\). The nodes pruned by !, r, ! (i.e., by its cutting sequence) are !, and its descendants. The cutting sequence of node ! is \((p), (q, !), (t, !), (!)\). The nodes pruned by node ! are r, r, ! and r.

Note that if a node \(Q\) is pruned by a cutting sequence \(D\) then \(Q\) does not precede in \(\text{seq}(T)\) any of the nodes of \(D\). (More precisely, if \(Q\) and the nodes of \(D\) occur in \(\text{seq}(T)\) then each node of \(D\) precedes \(Q\).) In [Apt97], the introducing node of (the cutting sequence of) an executing node \(!, \bar{A}\) is defined as the origin of the cut atom in \(!, \bar{A}\).

The definition above describes pruning due to a single cut in an LD-tree (or more generally—pruning trees in which no executing node is pruned). When there are more cuts, an executing node \(Q'\) may prune an executing node \(Q''\). Moreover, Examples 3.4, 3.6 show that in some cases some nodes pruned by \(Q''\) are not pruned by \(Q'\) and remain in the final tree. This should be considered while defining pruned LD-trees.

For the next definition we need to consider trees which are subgraphs of LD-trees. By a cutting sequence of a subgraph \(T'\) of an LD-tree \(T\) we mean a cutting sequence \(D\) of \(T\), such that each node of \(D\) is in \(T'\). By an executing node \(Q'\) of \(T'\) we mean a node of the form \(!, \bar{A}\).

**Definition 3.7** (pruned LD-tree) Let \(T\) be an LD-tree. Consider the possibly infinite sequence \(T\) of trees \(T_0, T_1, \ldots\), such that \(T_0 = T\) and

- \(T_i\) is obtained from \(T_{i-1}\) by removing the nodes pruned by the \(i\)th executing node in the sequence \(\text{seq}(T_{i-1})\) (for each \(T_i\) in \(T\), where \(i > 0\)),
- if some \(\text{seq}(T_n)\) (where \(n \geq 0\)) contains exactly \(n\) executing nodes then the sequence \(T\) is finite and \(T_n\) is its last element, otherwise \(T\) is infinite.

Let \(T'\) be the subgraph of \(T\) containing the nodes occurring in each of the trees \(T_0, T_1, \ldots\). (Thus if the sequence is finite then \(T'\) is its last element.) The pruned LD-tree \(\text{pruned}(T)\) corresponding to \(T\) (shortly: the pruned \(T\)) consists of those nodes of \(T'\) that occur in \(\text{seq}(T')\).

For informal explanation of the definition, consider the subgraph \(T''\) of \(T_{i-1}\) consisting of those nodes that are in \(\text{seq}(T_{i-1})\) between the root of \(T_{i-1}\) and \(Q_i\), the \(i\)th executing node in \(\text{seq}(T_{i-1})\). This subgraph describes the computation up to the \(i\)th executing node of the cut. Note that \(T''\) contains the cutting sequence of \(Q_i\). The nodes pruned by \(Q_i\) are absent from \(T_i\). Whole \(T''\) is a subgraph of \(T_i\). Also, \(T''\) will remain unchanged in (i.e., be a subgraph of) all the subsequent trees \(T_i, T_{i+1}, \ldots\).

**Example 3.8** Let \(T_0\) be the LD-tree from Example 3.4. The sequence \(\text{seq}(T_0)\) is: \((p), (q, !), (s, !, r, !), (!, r, !), (r, !), (t, !), (!), (r, r, !), r\). The sequence contains two executing nodes, the first one is \(!, r, !\). The tree \(T_1\) is \(T_0\) without the nodes pruned by \(!, r, !\), this means without \(t, !\) and its descendants (cf. Example 3.6). As there is only one executing node in \(\text{seq}(T_1)\), and all the nodes of \(T_1\) are in \(\text{seq}(T_1)\), the pruned tree \(\text{pruned}(T_0)\) is \(T_1\).

4. Correctness and completeness of programs

In preparation for the main subject of this work—program completeness related to the operational semantics with pruning, this section discusses some semantic issues abstracting from pruning. The purpose is twofold, introducing some concepts needed later on, and providing ground for comparing the proof methods based on declarative semantics, with the method of this paper, dealing with pruning. First we discuss correctness and completeness of programs, two notions related to the declarative semantics. We also present the standard ways of reasoning about program termination. Then we discuss a specific notion of correctness related to operational semantics, namely to LD-resolution. Note that we deal with semantic properties of programs; such properties are undecidable (as definite clause programs are Turing complete).
4.1. Declarative notions of correctness and completeness

Specifications From a declarative point of view, logic programs compute relations. A specification should describe these relations. It is convenient to assume that the relations are over the Herbrand universe. A handy way for describing such relations is a Herbrand interpretation; it describes, as needed, a relation for each predicate symbol of the program. So, by a specification we mean a Herbrand interpretation, i.e. a subset of \( \mathcal{HB} \).

Correctness and completeness In imperative and functional programming, (partial) correctness usually means that the program results are as specified (provided the program terminates). Logic programming involves non-determinism of a specific kind. A query may have 0, 1, or more answers, and the idea is to compute all of them. Thus in logic programming the notion of correctness divides into two: correctness (all the results are compatible with the specification) and completeness (all the results required by the specification are produced). In other words, correctness means that the relations defined by the program are subsets of the specified ones, and completeness means inclusion in the opposite direction. Formally:

Definition 4.1 Let \( P \) be a program and \( S \subseteq \mathcal{HB} \) a specification. \( P \) is correct w.r.t. \( S \) when \( \mathcal{M}_P \subseteq S \); it is complete w.r.t. \( S \) when \( \mathcal{M}_P \supseteq S \).

We will sometimes skip the specification when it is clear from the context. It is important to understand the relation between specifications and the answers of correct (or complete) programs [Dra16a]. A program \( P \) is correct w.r.t. a specification \( S \) iff, for any query \( Q \), \( Q \) being an answer of \( P \) implies \( S \models Q \). (Remember that \( Q \) is an answer of \( P \) iff \( P \models Q \).) Program \( P \) is complete w.r.t. \( S \) iff, for any ground query \( Q \), \( S \models Q \) implies that \( Q \) is an answer of \( P \). 5 For arbitrary queries, completeness of \( P \) w.r.t. \( S \) implies \( P \models Q \) when in the underlying alphabet there is a non-constant function symbol not occurring in \( P \), \( Q \), or there are \( k \) constants not occurring in \( P \), \( Q \), where \( k \geq 0 \) is the number of distinct variables occurring in \( Q \) [Dra16a]. In particular, the implication holds when the alphabet of function symbols is infinite (and \( P \) is finite) [Mah88]. (See [Dra16b] for further discussion.)

A note on pragmatics of the notion of completeness may be useful. Remember that the relations described by a specifications are on \( ground \) terms. So, strictly speaking, specifications do not describe program answers, but ground instances of the answers. For a non-ground \( Q \), it depends on the underlying alphabet whether \( S \models Q \) or not. 5 Informally, obtaining an answer \( A \in \mathcal{HB} \) from a computation (an SLD-tree) means that \( A \) is a ground instance of a computed answer of the tree. We are not interested whether \( A \) is actually a computed answer, or a more general computed answer has been produced. Similarly, obtaining answers \( A, A' \in \mathcal{HB} \) may happen when both of them are instances of a single computed answer, or they are (instances of) different computed ones.

Approximate specifications It happens quite often in practice that the relations defined by a program are not known exactly and, moreover, such knowledge is unnecessary. It is sufficient to specify the program’s semantics approximately. More formally, to provide distinct specifications, say \( S_{\text{compl}} \) and \( S_{\text{corr}} \), for completeness and correctness. The intention is that \( S_{\text{compl}} \subseteq \mathcal{M}_P \subseteq S_{\text{corr}} \), where \( \mathcal{M}_P \) is the least Herbrand model of the program. So the specification for completeness says what the program has to compute, and the specification for correctness—what it may compute. In other words, the program should not produce any answers rejected by the specification for correctness. It is irrelevant whether atoms from \( S_{\text{corr}} \setminus S_{\text{compl}} \) are, or are not, answers of the program. Various versions of the program may have different semantics, but each version should be correct w.r.t. \( S_{\text{corr}} \) and complete w.r.t. \( S_{\text{compl}} \). As an example, consider the standard append program, and atom \( A = \text{append}([a, \ldots, [a|1]]) \). It is irrelevant whether \( A \) is an answer of the program, or not. For further discussion and examples see [Dra16a, DM05], see also Example 4.6.

Reasoning about correctness Although it is outside of the scope of this paper, we briefly mention proving program correctness. A sufficient condition for a program \( P \) being correct w.r.t. a specification \( S \) is \( S \models P \). In other words, for each ground instance \( H \leftarrow B_1, \ldots, B_n \) of a clause of \( P \), if \( B_1, \ldots, B_n \in S \cup \{!\} \) then \( H \in S \). Deransart [Der93] attributes this method to [Cla79]. See [Dra16a, DM05] for examples and discussion.

---

5 To show that the equivalence does not hold in general, assume a two element alphabet of function symbols, with a unary \( f \) and a constant \( a \). Take \( P = \{ p(a), p(f(X)) \} \), \( S = \mathcal{HB} \), \( Q = p(X) \). The program is complete w.r.t. \( S \) and \( S \models Q \), but \( Q \) is not an answer of \( P \).

6 In the example from footnote 5 we have \( S \models p(X) \), but adding a constant \( b \) to the alphabet (and keeping \( S \) unchanged) results in \( S \not\models p(X) \), as \( S \not\models p(b) \).
Reasoning about completeness  Little work has been devoted to reasoning about completeness of programs. See [Dra16a] for an overview. We summarize the approach from [Dra16a], also presented in [Dra15]. That approach is a starting point for the method introduced in this paper. We first need two auxiliary notions.

Definition 4.2 A program $P$ is complete for a query $Q$ w.r.t. $S$ when $S \models Q\theta$ implies that $Q\theta$ is an answer for $P$, for any ground instance $Q\theta$ of $Q$.

Informally, $P$ is complete for $Q$ when all the ground answers for $Q$ required by the specification $S$ are answers of $P$. Note that a program is complete w.r.t. $S$ iff it is complete w.r.t. $S$ for any query $A \in S$.

Definition 4.3 A program $P$ is semi-complete w.r.t. a specification $S$ if $P$ is complete w.r.t. $S$ for any query $Q$ for which there exists a finite SLD-tree.

Less formally, the existence of a finite SLD-tree means that, under some selection rule, the computation for $Q$ and $P$ terminates. (Sometimes this is called “universal termination”). For a semi-complete program $P$, if the computation terminates then all the answers for $Q$ required by the specification have been obtained. In other words, $P$ is complete for query $Q$. So establishing completeness may be done in two steps: showing semi-completeness and termination. Obviously, a complete program is semi-complete.

Our sufficient condition for semi-completeness employs the following notion, stemming from [Sha83].

Definition 4.4 A ground atom $H$ is covered by a clause $C$ w.r.t. a specification $S$ if $H$ is the head of a ground instance $H \leftarrow B_1, \ldots, B_n (n \geq 0)$ of $C$, such that all the atoms $B_1, \ldots, B_n$ are in $S \cup \{1\}$.

A ground atom $H$ is covered by a program $P$ w.r.t. $S$ if it is covered w.r.t. $S$ by some clause $C \in P$.

Theorem 4.5 (semi-completeness [Dra16a]) If all the atoms from a specification $S$ are covered w.r.t. $S$ by a program $P$ then $P$ is semi-complete w.r.t. $S$.

Example 4.6 Consider the well-known program APPEND:

$$app([], L, L). \quad \text{app}([H|K], L, [H|M]) \leftarrow \text{app}(K, L, M).$$

and a specification

$$S^0_{\text{APPEND}} = \{ app(k, l, m) \in \mathcal{HB} \mid k, l, m \text{ are lists, } k \ast l = m \},$$

where $k \ast l$ stands for the concatenation of lists $k$ and $l$. Consider an atom $A = \text{app}(k, l, m) \in S^0_{\text{APPEND}}$. If $k = []$ then $A = \text{app}([], l, l)$ and $A$ is covered by the first clause of the program. Otherwise $A = \text{app}([h|k'], l, [h|m'])$, where $k' \ast l = m'$. Thus $A$ is covered by an instance $\text{app}([h|k'], l, [h|m']) \leftarrow \text{app}(k', l, m')$ of the second clause of APPEND. Hence by Theorem 4.5 APPEND is semi-complete w.r.t. $S^0_{\text{APPEND}}$. It is also complete w.r.t. $S^0_{\text{APPEND}}$ as it terminates for each $A \in S^0_{\text{APPEND}}$ (we skip a simple proof [Dra16a] that the program is recurrent, see Theorem 4.8 below). Note that APPEND is not correct w.r.t. $S^0_{\text{APPEND}}$, as it has answers whose arguments are not all lists, e.g. $\text{app}([a], 1, [a|1])$. (The program is correct w.r.t. specification $\{ app(k, l, m) \in \mathcal{HB} \mid \text{if } l \text{ or } m \text{ is a list then } app(k, l, m) \in S^0_{\text{APPEND}} \}$ [DM05, Dra16a].)

4.2. Reasoning about termination

Termination—this means finiteness of (S)LD-trees—is needed to conclude completeness from semi-completeness, and will also be needed for the main result of this paper. We now briefly summarize basic approaches to proving program termination [Apt97].

Definition 4.7 A level mapping is a function $| |: \mathcal{HB} \to \mathbb{N}$ assigning natural numbers to ground atoms. We additionally assume that $|[]| = 0$.

A program $P$ is recurrent w.r.t. a level mapping $| |$ [Bez93, Apt97] if, in every ground instance $H \leftarrow B_1, \ldots, B_n \in \text{ground}(P)$ of its clause ($n \geq 0$), $|H| > |B_i|$ for all $i = 1, \ldots, n$. A program is recurrent if it is recurrent w.r.t. some level mapping.

A program $P$ is acceptable w.r.t. a specification $S$ and a level mapping $| |$ if $P$ is correct w.r.t. $S$, and for every $H \leftarrow B_1, \ldots, B_n \in \text{ground}(P)$ we have $|H| > |B_i|$ whenever $S \models B_1, \ldots, B_{i-1}$. A program is acceptable if it is acceptable w.r.t. some level mapping and some specification.
A query $Q$ is bounded w.r.t. a level mapping $| |$ if there exists a $k \in \mathbb{N}$ such that for each ground instance $A$ of an atom of $Q$ we have $|A| < k$.

The definition of acceptable is more general than that of [AP93, Apt97], which additionally requires $S$ to be a model of $P$. Both definitions make the same programs acceptable [Dra16a].

**Theorem 4.8** (termination [Bez93, AP93]) If $P$ is a recurrent program and $Q$ a bounded query then all SLD-derivations for $P$ and $Q$ are finite.

If a program $P$ is acceptable w.r.t. some specification and some level mapping then all LD-derivations for $P$ and a bounded query $Q$ are finite.

Hence each SLD-tree for $P$, $Q$ in the first case, and the LD-tree for $P$, $Q$ in the second case is finite (as programs are finite). The second implication of the theorem holds for a more general class of queries (bounded w.r.t. $S$) [Apt97]; we skip the details. It follows that if a (finite) program $P$ is (i) semi-complete w.r.t. a specification $S$ and (ii) recurrent or acceptable w.r.t. some level mapping (and some specification $S'$) then $P$ is complete w.r.t. $S$.

### 4.3. A notion of correctness related to operational semantics

The subject of this paper is completeness of programs when the search space is pruned by means of the cut. Such operational semantics does not preserve some basic properties of SLD-resolution. For example an instance of a query $Q$ may succeed while $Q$ fails (e.g. consider program $p(a) :- !, . p(X)$, for which query $p(X)$ fails and $p(b)$ succeeds). Also, we need to reason about which atoms are selected in derivations. So a declarative approach is no longer possible; we have to reason in terms of the operational semantics, in other words, to express and prove properties inexpressible in terms of specifications, correctness, and completeness of Sect. 4.1.

This section presents a method of reasoning about which atoms are selected in LD-derivations, and which atoms are computed answers (of subderivations). The approach stems from [DM88] and is due to [BC89], we prove properties inexpressible in terms of specifications, correctness, and completeness of Sect. 4.1.

Specifications of another kind are needed here, let us call them c-s specifications (c-s for call-success).

**Definition 4.9** (c-s-correctness) A c-s-specification is a pair $pre, post$ of sets of atoms, closed under substitution. The sets $pre, post \subseteq TB$ are called, respectively, precondition and postcondition.

A program is c-s-correct w.r.t. a c-s-specification $pre, post$ when in each LD-derivation $D$ every selected atom is in $pre \cup \{!\}$, and each atomic computed answer (of a successful subderivation of $D$) is in $post \cup \{!\}$, provided that $D$ begins with an atomic query from $pre$.

For c-s-correct programs with more general initial queries, see below. The notion of c-s-correctness will be employed in the main part of this work.

**Definition 4.10** (well-asserted) Let $pre, post$ be a c-s-specification. A clause $C$ is well-asserted (w.r.t. $pre, post$) if for each (possibly non-ground) instance $H \leftarrow B_1, \ldots, B_n$ of $C$ ($n \geq 0$)

if $H \in pre$ and $B_1, \ldots, B_k \in post \cup \{!\}$ then $B_{k+1} \in pre \cup \{!\}$, for $k = 0, \ldots, n - 1$,

if $H \in pre$ and $B_1, \ldots, B_n \in post \cup \{!\}$ then $H \in post$.

A program is well-asserted if so are all its clauses.

A query $Q$ is well-asserted (w.r.t. $pre, post$) when the clause $p \leftarrow Q$ is well-asserted w.r.t. $pre \cup \{p\}, post \cup \{p\}$, where $p \in HB$ is a predicate symbol not occurring in $P, pre, post$.

Note that the first atom of a well-asserted query is in $pre \cup \{!\}$, and that if all atoms of a query $Q$ are in $pre \cup \{!\}$ then $Q$ is well-asserted.

The following sufficient condition follows from Corollaries 8.8 and 8.9 of [Apt97] (with an obvious generalization to programs with the cut).

**Theorem 4.11** (c-s-correctness) Let $P$ be a program and $pre, post$ a c-s-specification. If $P$ is well asserted w.r.t. $pre, post$ then $P$ is c-s-correct w.r.t. $pre, post$. 

The definition of c-s-correctness involves only atomic initial queries. For general queries, consider a c-s-specification \( \text{pre}, \text{post} \), a program \( P \), and a query \( Q \). If \( P \) is c-s-correct, and \( Q \) is well-asserted then in each LD-derivation \( D \) for \( P \) and \( Q \) each selected atom is in \( \text{pre} \cup \{ ! \} \) and each atomic computed answer (of a subderivation) is in \( \text{post} \cup \{ ! \} \). More generally, each atom of the computed answer of a subderivation of \( D \) is in \( \text{post} \cup \{ ! \} \), as \( \text{post} \) is closed under substitution.

5. Completeness in the presence of the cut

This section introduces a sufficient condition for completeness of pruned LD-trees. The main result is preceded by some necessary definitions.

**Definition 5.1** Let \( T \) be an LD-tree, or a pruned LD-tree, and \( Q \) be its root. An answer of \( T \) is the computed answer of a successful LD-derivation which is a branch of \( T \).

The tree \( T \) is complete w.r.t. a specification \( S \subseteq \mathcal{HB} \) if, for any ground \( \theta, S \models \theta \) implies that \( \theta \) is an instance of an answer of \( T \).

Informally, \( T \) is complete iff it produces all the answers for its root which are required by \( S \).

The next definition is, in a sense, the main part of our sufficient condition for completeness. The idea is to require not only that a ground atom \( A \) is covered by a clause \( C \), but also that the tree node introduced by \( C \) cannot be pruned by a cut in a preceding clause. Moreover, if the cut is present in \( C \), say \( C = H \leftarrow \bar{B}_0, !, \bar{B}_1 \), then \( A \) should be produced by \( C \) employing an arbitrary answer to \( \bar{B}_0 \). To formalize this idea it is necessary to employ a c-s-specification, to describe the atoms possibly selected in the LD-trees and the corresponding computed answers.

**Definition 5.2** Let \( S \subseteq \mathcal{HB} \) be a specification, and \( \text{pre}, \text{post} \) a c-s-specification (\( \text{pre}, \text{post} \in \mathcal{T}B \)). A ground atom \( A \) is c-covered (contextually covered) w.r.t. \( S \) and \( \text{pre}, \text{post} \) by a clause \( C \) occurring in a program \( P \) if

1. \( A \) is covered by \( C \) w.r.t. \( S \), and
2. if \( C = H \leftarrow \ldots \) is preceded in \( P \) by a clause \( C' = H' \leftarrow \bar{A}_0, !, \bar{A}_1 \), where both \( H \) and \( H' \) have the same predicate symbol, and \( ! \) does not occur in \( \bar{A}_0 \), then
   - for any atom \( H'' \in \text{pre} \) such that \( A \) is an instance of \( H'' \)
   - no ground instance \( H'' \theta \) of \( H'' \) is covered by \( H' \leftarrow \bar{A}_0 \) w.r.t. \( \text{post} \cap \mathcal{HB} \);
3. if \( C \) contains the cut, \( C = H \leftarrow \bar{B}_0, !, \bar{B}_1 \), then
   - for any instance \( H \rho \in \text{pre} \) such that \( A \) is an instance of \( H \rho \) (and \( \rho \) is as below),
   - for any ground instance \( \bar{B}_0 \eta \) such that \( \bar{B}_0 \eta \subseteq \text{post} \cup \{ ! \} \) (and \( \eta \) is as below),
   - \( A \) is covered by \( (H \leftarrow \bar{B}_1)\eta \) w.r.t. \( S \),

where \( \text{dom}(\rho) \subseteq \text{vars}(H), \text{rng}(\rho) \cap \text{vars}(C) \subseteq \text{vars}(H), \text{dom}(\rho) \cap \text{rng}(\rho) = \emptyset \), and \( \text{dom}(\eta) = \text{vars}(\bar{B}_0 \rho) \).

We say that \( A \) is c-covered (w.r.t. \( S \) and \( \text{pre}, \text{post} \)) by a program \( P \) if it is c-covered (w.r.t. \( S \) and \( \text{pre}, \text{post} \)) by a clause from \( P \). Similarly, \( S \) is c-covered by \( P \) if each atom from \( S \) is c-covered by \( P \).

Some informal explanation may be useful. The role of condition 2 is to exclude cases where for query \( H'' \) the cut in a clause \( C' \) preceding \( C \) is executed, which results in not applying \( C \) for \( H'' \). Roughly speaking, the cut in \( C' = H' \leftarrow \bar{A}_0, !, \bar{A}_1 \) is executed when \( \bar{A}_0 \) succeeds. What we know about the computed answer for \( \bar{A}_0 \) obtained at the success is that each atom of the answer is in \( \text{post} \). So the cut in \( C' \) may be executed if there is an instance \( (H' \leftarrow \bar{A}_0)\varphi \), such that its head \( H' \varphi \) is an instance of \( H'' \) and \( \bar{A}_0 \varphi \subseteq \text{post} \). It is sufficient here to consider only ground instances of \( H' \leftarrow \bar{A}_0 \), such a ground instance exists if a ground instance of \( H'' \) is covered by \( H' \leftarrow \bar{A}_0 \) w.r.t. \( \text{post} \cap \mathcal{HB} \). When the cut is executed in clause \( C = H \leftarrow \bar{B}_0, !, \bar{B}_1 \), only the first answer for \( \bar{B}_0 \) will be used. The only information we have about this answer is that its atoms are in \( \text{post} \cup \{ ! \} \). The role of condition 3 is to ensure that for each such answer clause \( C \) can produce \( A \). To ensure that such answer exists, \( C \) is required to cover \( A \) w.r.t. \( S \).
For programs without the cut, $c$-covered is equivalent to covered. For multiple occurrences of the cut in a clause, condition 2 considers the first occurrence of $!$ in $C'$, while what matters in condition 3 is the last occurrence of $!$ in $C$ (if the condition holds for the last occurrence then it holds for each previous one). The two conditions get simplified when all the atoms of $pre$ are ground, as then $H'' = A = H''θ$ and $Hρ = A$. In a general case, checking that an atom is $c$-covered by a clause can be simplified as follows:

**Remark 5.3** Note that in condition 2 of Definition 5.2, instead of considering all atoms $H'' ∈ pre$, it is sufficient to consider maximally general atoms $H'' ∈ pre$ unifiable with $H'$ (and having $A$ as an instance).

Similarly, by Proposition A.7 in the Appendix, instead of considering all instances $Hρ ∈ pre$ of $H$ in condition 3, it is sufficient to consider maximally general instances $Hρ ∈ pre$.

Assume that $S ⊆ post$, and that condition 3 holds. Then, for $A$ to be is covered by a clause $C = H ← B₀, !, B₁$ w.r.t. $S$ (condition 1), it is sufficient that $A$ is covered by $H ← B₀$ w.r.t. $S$.

The core of the proposed method of proving completeness is the following sufficient condition.

**Theorem 5.4 (completeness)** Consider an LD-tree $T$ for a program $P$, and the tree $pruned(T)$. Let $Q$ be the root of $T$. Assume that $Q$ does not contain $. Let $S ⊆ ᴨB$ be a specification, and $pre, post$ a $c$-$s$-specification such that $S ⊆ post$. Let

- $pruned(T)$ be finite, $P$ be $c$-$s$-correct w.r.t. $pre, post$, $Q$ be a well asserted query w.r.t. $pre, post$, and
- each $A ∈ S$ be $c$-covered w.r.t. $S$ and $pre, post$ by a clause of $P$.

Then $pruned(T)$ is complete w.r.t. $S$.

The proof is presented in the Appendix. The next section contains example completeness proofs employing this theorem.

**Additional comments** These remarks may be skipped at the first reading.

To deal with an initial query $Q$ containing the cut, one may add a clause $p(\vec{V}) ← Q$ to the program (where $\vec{V}$ are the variables of $Q$, and $p$ is a new symbol), and extend the specifications appropriately. Specification $S$ should be extended to $S' = S ∪ \{ p(\vec{V})θ ∈ ᴨB \mid Qθ ⊆ S ∪ \{!\} \}$, and all the $p$-atoms should be added to $pre$ and to $post$. Then Theorem 5.4 is applicable to the extended program. (Note that clause $p(\vec{V}) ← Q$ covers each $p$-atom of $S'$ w.r.t. $S'$, by the definition of $S'$, and that condition 2 of Definition 5.2 vacuously holds for $p(\vec{V}) ← Q$. Note also that $p(\vec{V}) ← Q$ satisfies the sufficient condition for $c(s)$-correctness, i.e. is well-asserted w.r.t. the new $c$-$s$-specification, as $Q$ is well-asserted w.r.t. $pre, post$.) We skip further details.

The theorem is inapplicable to infinite pruned trees. This restriction is not easy to overcome: the proof of the theorem is based on constructing a non-failing branch of the tree, the branch—if finite—provides the required answer for $Q$.

We would like to note a technical detail: Definition 5.2 actually refers to $post ∩ ᴨB$, not to the whole $post$.

A version of Definition 5.2 is possible, which instead of $post$ employs a specification $S^+ ∈ ᴨB$ w.r.t. which the program is correct; $post$ is replaced by $S^+$ in conditions 2, 3. (So in this version, clauses are $c$-covered w.r.t. $S$, and $pre, S^+$.) We state without proof that Theorem 5.4 (with obvious modifications) also holds for such modified notion of $c$-covered. (The modifications are: requiring that $P$ is correct w.r.t. $S^+$, and $c$-$s$-correct w.r.t. $pre, post'$, for some $post'$; all other occurrences of $post$ are replaced by $S^+$.) In this way a $c$-$s$-specification is used only to describe the atoms selected in the derivations, and the specification $S^+$ describes the obtained computed answers. We expect that such separation may be convenient in some cases, e.g. when correctness of $P$ w.r.t. $S^+$ has already been established.

Theorem 5.4 provides a sound method for proving completeness of pruned trees. We only briefly discuss the issue of (in)completeness of the method. For examples for which the sufficient condition from the theorem is not a necessary one, see Example 6.4 in the next section. See [Dra16a] for a discussion of the same issue in a related (but simpler) case of proving completeness of logic programs. Here we only mention a specific reason of incompleteness of the proposed method: Speaking informally, in condition 3 of Definition 5.2 it would be sufficient to consider the first answer to $B₀ρ$. Instead we consider all possible answers described by $post$ (as we do not know the ordering of the actual answers).
6. Examples

This section presents three example proofs of completeness of pruned trees. The first one considers a case where various branches produce the same answer and some of them are pruned. The second is a rather artificial example, to illustrate some details of Definition 5.2. In the third example we prove completeness of the usual way of implementing negation as failure in Prolog, provided the initial query is ground.

Example 6.1 Consider a program IN (which describes the subset relation between the sets of elements of two ground lists):

\[
\begin{align*}
\text{in}([], L), & \quad \text{in}(E|T], L) := m(E, L), !, \text{in}(T, L). \\
m(E, [E|L]), & \quad m(E, [H|L]) := m(E, L).
\end{align*}
\]

In the program, a single answer for \(m(E, L)\) is sufficient (to obtain the required answer for a ground in-atom). So the cut is used to prune further answers for \(m(E, L)\). Consider specifications

\[
S = S_m \cup S_{in}, \quad \text{pre} = \text{pre}_{m} \cup \text{pre}_{in}, \quad \text{post} = TB,
\]

where

\[
\begin{align*}
S_m &= \{ m(t_i, [t_1, \ldots, t_n]) \in HB \mid 1 \leq i \leq n \}, \\
S_{in} &= \{ \text{in}([u_1, \ldots, u_m], [t_1, \ldots, t_n]) \in HB \mid m, n \geq 0, \{ u_1, \ldots, u_m \} \subseteq \{ t_1, \ldots, t_n \} \}, \\
\text{pre}_{m} &= \{ m(u, t) \in TB \mid t \text{ is a list} \}, \\
\text{pre}_{in} &= \{ \text{in}(u, t) \in HB \mid u, t \text{ are ground lists} \}.
\end{align*}
\]

The program is c-s-correct w.r.t. pre, post (by Theorem 4.11, we skip rather simple details). We show that each atom \(A = \text{in}(u, t) \in S_{in}, \) where \(u = [u_1, \ldots, u_m], m > 0, \) is c-covered by the second clause \(C \) of IN. Note first that \(A \) is covered by \(C, \) due to its instance \(\text{in}([u_1[|u_2, \ldots, u_m]], t) := m(u_1, t), !, \text{in}([u_2, \ldots, u_m], t); \) its body atoms are in \(S, \) as each \(u_i \) is a member of \(t. \) Condition 2 of Definition 5.2 holds trivially, as no ! occurs in the preceding clause.

To check condition 3, take an instance \(\text{in}(E|T], L)\rho \in \text{pre} \) of the head of \(C. \) The instance is ground, and the whole \(C\rho \) is ground. So in Definition 5.2, \(\rho \eta = \rho. \) If \(A \) is an instance of (thus equal to) \(\text{in}(E|T], L)\rho \) then \(\text{in}(T, L)\rho = \text{in}([u_2, \ldots, u_m], t) \in S \) (as \(A \in S \)). Thus \(A \) is covered by \(\text{in}(E|T], L) := \text{in}(T, L)\rho \eta. \) So condition 3 of Definition 5.2 holds. Thus \(A \) is c-covered by \(C. \) It is easy to check that all the remaining atoms of \(S \) are covered and c-covered w.r.t. \(S \) by the remaining clauses of IN.

Note that program IN is recurrent under the level mapping \(|m(s, t)| = |t|, |\text{in}(s, t)| = |s| + |t|, \) where \(|[t]| = 1 + |t| \) and \([f(t_1, \ldots, t_n)] = 0 \) (for any ground terms \(h_1, t_1, \ldots, h_n, \) and any function symbol \(f \) distinct from \([[]]. \) Thus each LD-tree for IN and a query \(Q \in \text{pre} \) is finite.

By Theorem 5.4, for each \(Q \in \text{pre} \) the pruned LD-tree is complete w.r.t. \(S. \) Notice that condition 3 may not hold when non ground arguments of \(\text{in} \) are allowed in \(\text{pre}_{in}, \) and that for such queries the pruned LD-trees may be not complete w.r.t. \(S. \) As an example take \(H' = \text{in}([X], [1, 2]) \) and \(A = \text{in}([2], [1, 2]). \)

The previous example illustrates a practical case of so called “green cut” [SS94], where (for certain queries) pruning does not remove any answers. However it represents a rather simple application of Theorem 5.4, with an easy check for condition 2 of Definition 5.2, and condition 3 applied only to ground atoms from \(\text{pre}. \) The next two examples illustrate more sophisticated cases of conditions 2, 3.

Example 6.2 Consider a program \(P:\)

\[
\begin{align*}
p(X, Y) & := q(X, Y), r(X, Y), !. \\
p(X, Z) & := q(X, Y), !, r(Y, Z).
\end{align*}
\]

and specifications

\[
\begin{align*}
S &= \{ p(a, c), q(a, a'), r(a, c), r(a', c) \}, \\
\text{post} &= S \cup \{ q(a, a) \}, \\
\text{pre} &= \{ p(a, t) \mid t \in T \cup \{ q(a, t) \mid t \in T \} \cup \{ r(t, u) \mid t, u \in T \}. \}
\end{align*}
\]
The program is c-s-correct w.r.t. \( \text{pre}, \text{post} \), by Theorem 4.11. \(^7\) We show that atom \( A = p(a, c) \in S \) is c-covered by the second clause of \( P \). Note first that \( A \) is covered by the clause w.r.t. \( S \) due to its instance \( p(a, c) : \sim q(a, a') \). \(^8\)

For condition 2 of Definition 5.2 it is sufficient to consider \( H'' = p(a, X) \), by Remark 5.3. No ground instance \( p(a, s) \) of \( H'' (s \in \mathcal{H}(U)) \) is covered by \( p(X, Y), r(X, Y) \) w.r.t. \( \text{post} \cap \mathcal{HB} \), as in no ground instance of \( q(X, Y), r(X, Y) \) both atoms are in \( \text{post} \). So condition 2 holds.

By Remark 5.3, it is sufficient to check condition 3 of Definition 5.2 for \( \rho = \{X/a\} \), as \( p(X, Z)\rho = p(a, Z) \) is a most general \( p \)-atom in \( \text{pre} \). If \( q(X, Y)\rho \eta \in \text{post} \) (and \( \text{dom}(\eta) = \text{vars}(q(X, Y)\rho) = \{Y\} \)) then \( \eta = \{Y/a\} \) or \( \eta = \{Y/a'\} \). Hence \( r(Y, Z)\rho \eta \) is \( r(a, Z) \) or \( r(a', Z) \). In both cases, \( p(a, c) : \sim r(Y, \eta, c) \) is a ground instance of \( p(X, Z) : \sim r(Y, Z) \rho \eta \) (i.e. of \( p(a, Z) : \sim r(Y, \eta, Z) \)) covering \( p(a, c) \) w.r.t. \( S \). So condition 3 holds, and \( p(a, c) \) is covered by \( P \).

The remaining atoms of \( S \) are trivially c-covered by the unary clauses of \( P \). For any \( A \in \text{pre} \), the LD-tree for \( P \) and \( A \) is finite, hence the pruned LD-tree is complete w.r.t. \( S \) by Theorem 5.4.

Example 6.3 This example deals with a standard implementation of negation as finite failure by means of the cut. We show that the negation of a ground \( p(t) \) succeeds whenever \( p(t) \) finitely fails. Consider a program \( P_0 \) without the cut. Assume that \( P_0 \) is c-s-correct w.r.t. a specification \( \text{pre}_0, \text{post}_0 \), and that predicate symbols \( \text{notp}, \text{fail} \) do not occur in \( P_0, \text{pre}_0, \text{post}_0 \). Let \( p \) be a unary predicate symbol. Let \( P \) be \( P_0 \) with the following clauses added:

\[
\text{notp}(X) \leftarrow \text{p}(X), \text{fail}.
\]

Let \( \text{pre}_{\text{notp}} = \{ \text{notp}(t) \in \mathcal{HB} \mid p(t) \in \text{pre}_0 \} \). Program \( P \) is c-s-correct w.r.t. the c-s-specification \( \text{pre}, \text{post} \), where \( \text{pre} = \text{pre}_0 \cup \text{pre}_{\text{notp}} \cup \{ \text{fail} \} \), and \( \text{post} = \text{post}_0 \cup \text{pre}_{\text{notp}} \). We show that \( \text{notp}(t) \) succeeds for those ground \( t \) for which \( p(t) \) is known to finitely fail (i.e. the LD-tree for \( p(t) \) and \( P_0 \) is finite, and \( p(t) \nsubseteq \text{post}_0 \), hence \( p(t) \) does not succeed). Formally, the property to be proven is that the finite pruned LD-trees for \( P \) are complete w.r.t. a specification \( S = \{ \text{notp}(t) \in \text{pre}_{\text{notp}} \mid p(t) \nsubseteq \text{post}_0 \} \cup \mathcal{M}_{P_0} \).

Take an atom \( A = \text{notp}(t) \in S \). To show that \( A \) is c-covered w.r.t. \( S, \text{pre}, \text{post} \) by clause \( \text{notp}(X) \) of \( P \), condition 2 of Definition 5.2 and the clause \( C' = \text{notp}(X) : \sim p(X), \text{fail} \) have to be considered. Using the notation of Definition 5.2, \( H'' = A, H'' \) is ground, and \( H'' \) is not covered by \( \text{notp}(X) : \sim p(X) \) w.r.t. \( \text{post} \cap \mathcal{HB} \) (as \( p(t) \nsubseteq \text{post} \)). Thus condition 2 holds and \( A \) is c-covered by a clause of \( P \). The remaining atoms of \( S \) (those from \( \mathcal{M}_{P_0} \)) are obviously covered by the clauses of \( P_0 \), as no cut occurs in \( P_0 \), and each atom of \( \mathcal{M}_{P_0} \) is covered by \( P_0 \) w.r.t. \( \mathcal{M}_{P_0} \). By Theorem 5.4 each finite pruned LD-tree for \( P \) with an atomic root from \( \text{pre} \) is complete w.r.t. \( S \).

Note that condition 2 may not hold when \( \text{pre} \) contains non-ground \( \text{notp} \)-atoms, and that for such query the pruned LD-tree may not be complete w.r.t. \( S \). (Assume that \( \text{pre} \) contains an atom \( B = \text{notp}(u) \) such that (i) \( p(u) \) succeeds, but (ii) \( p(u) \nsubseteq \text{post} \) for some ground instance \( u \sigma \in \mathcal{H}(U) \) of \( u \). Hence \( B \sigma \) fails, by (i), and \( B \sigma \in S \), by (ii). So the pruned LD-tree for \( B \) is not complete w.r.t. \( S \). On the other hand, \( B \sigma \) is not c-covered by \( P \) w.r.t. \( S, \text{pre}, \text{post} \), as condition 2 is violated. Take \( H'' = B \), and a ground instance \( p(u \theta) \in \mathcal{HB} \) of the answer for \( p(u) \).

Thus \( p(u \theta) \in \text{post} \cap \mathcal{HB} \). So a ground instance \( \text{notp}(u \theta) \) of \( H'' \) is covered by \( \text{notp}(X) : \sim p(X) \text{ w.r.t. } \mathcal{HB} \), which is forbidden by condition 2.)

Example 6.4 Here we show incompleteness of the proposed method, i.e. show that the sufficient condition of 5.4 is not a necessary one. Consider program \( P \), specification \( S \), and c-s-specification \( \text{pre}, \text{post} \) from Example 6.2. Let us replace \( \text{post} \) by \( \text{post}' \) as follows. Obviously, \( P \) remains c-s-correct w.r.t. \( \text{pre}, \text{post}' \). Now condition 3 of Definition 5.2 is violated for \( S, \text{pre}, \text{post}' \), as it is possible that \( \rho = \{X/a\} \) and \( \eta = \{Y/b\} \), thus \( (H \leftarrow \bar{B})\eta \) is \( p(a, Z) \leftarrow r(b, Z) \), which has no instance covering \( p(a, c) \) w.r.t. \( S \). However the pruned tree for any \( A \in \text{pre} \) is complete, as shown in Example 6.2. (Let us also note that replacing \( \text{post} \) by \( \mathcal{HB} \) results in violating conditions 2 and 3.)

Similarly, let us replace specification \( S \) by \( S' = \{ p(a, c) \} \). Note that \( S' \subseteq S \). Obviously, the pruned LD-trees shown complete w.r.t. \( S \) are complete w.r.t. \( S' \). However the conditions of Definition 5.2 do not hold for \( S', \text{pre}, \text{post} \), as condition 1 of Definition 5.2 is violated (atom \( p(a, c) \) is not covered by \( P \) w.r.t. \( S' \)).

\(^7\) Note that \( S \) does not require \( q(a, a) \) or \( q(b, b) \) to be computed, and that \( P \) is not correct w.r.t. \( S \), cf. “Approximate specifications” in Sect. 4.1. Note also a usual situation: even if we are interested in completeness w.r.t. \( \{p(a, c)\} \), some \( q \)- and \( r \)-atoms are to be present in \( S \) in order to facilitate the proof. For further discussion see Example 6.4.

\(^8\) Note also that each atom of \( S \) is covered by \( P \) w.r.t. \( S \), and that \( P \) is recurrent. Thus \( P \) is complete w.r.t. \( S \) (by Theorem 4.5 and the remark following Theorem 4.8).
The two cases above show that to apply Theorem 5.4 we sometimes need to weaken $S$, or strengthen $post$. Now we show a case where such simple modifications do not help. Let us replace $pre$ by $TU$, and $post$ by $post'' = post \cup \{q(b, b)\}$. Note that the pruned LD-trees for $P$ and any $A \in TU$ are complete w.r.t. $S$. We show that this fact cannot be obtained by Theorem 5.4, as the sufficient condition of the theorem does not depend on the order of clauses in procedure $q$, and changing this order may destroy completeness.

Let $P'$ be $P$ with clause $q(b, b)$ moved to be the first clause of procedure $q$. The pruned LD-tree for $P'$ and $p(X, Z)$ does not contain any answer, so it is not complete w.r.t. $S$. Thus the premises of Theorem 5.4 do not hold. As both $P$ and $P'$ are c-s-correct w.r.t. $TU$, $post''$, the conditions of Definition 5.2 do not hold for $S$ and $TU$, $post''$. (Indeed, in condition 3, $(H \leftarrow B_1)p\eta$ may be $p(b, Z) \leftarrow r(b, Z)$; this clause does not cover $p(a, c)$.)

7. Conclusion

This paper introduces a sufficient condition for completeness of Prolog programs with the cut. The syntax is formalized as definite clause programs with the cut. The operational semantics is formalized in two steps: LD-resolution, and pruning LD-trees. The sufficient condition is illustrated by example completeness proofs.

Acknowledgements

Thanks are due to Paul Tarau for noticing an incorrect claim in the first version of this work. The author thanks anonymous reviewers for the pinpointed errors and for suggestions concerning the presentation.

A. Appendix

The appendix contains a proof of Theorem 5.4. It also introduces a proposition (A.7) employed in Remark 5.3 (to simplify checking condition 3 of Definition 5.2). The proof employs the notions of unrestricted derivation and lift from [Doe94, Definitions 5.9, 5.35]. We first present their definitions and the main related technical result, adjusted to the Prolog selection rule and to the difference (explained in Sect. 3.1) between the standard notion of SLD-derivation adopted here and the notion of derivation of [Doe94].

Definition A.1 An unrestricted LD-derivation for a program $P$ is a (finite or infinite) sequence $Q_0, Q_1, \ldots$ of queries, together with a sequence $\theta_1, \theta_2, \ldots$ of substitutions (called specializations) and a sequence $C_1, C_2, \ldots$ of clauses from $P$, such that for each $Q_i$ ($i \neq 0$):

- if $Q_{i-1} = !$, $\bar{A}$ then $Q_i = \bar{A}$ and $\theta_i = \epsilon$, otherwise
- if $Q_{i-1} = A$, $\bar{A}$ then $Q_i = \bar{B}$, $A\theta_i$, where $dom(\theta_i) \subseteq vars(Q_{i-1})$ and $A\theta_i \leftarrow \bar{B}$ is an instance of $C_i$.

An unrestricted LD-derivation is successful if its last query is empty. The answer of such successful derivation $Q_0, \ldots, Q_n; \theta_1, \ldots, \theta_n; C_1, \ldots, C_n$ is $Q_0\theta_1 \cdots \theta_n$.

So, informally, the difference between derivations and unrestricted derivations is that the latter employ clause instances which may be not most general ones. A technical difference is that a most general unifier in a derivation applies both to the variables of a query and those of a clause (variant), while in an unrestricted derivation a specialization applies only to the variables of a query. Moreover, a specialization $\theta_i$ may replace (by a non variable) a variable of $Q_{i-1}$ which does not occur in the first atom of $Q_{i-1}$.

When presenting (unrestricted) derivations, we sometimes skip the sequence of clauses, or the sequence of substitutions. Note that if $Q_0, Q_1, \ldots; \theta_1, \theta_2, \ldots$ is an LD-derivation then $Q_0, Q_1, \ldots; \theta_1 \mid Q_0, \theta_2 \mid Q_0, \ldots$ (together with a suitable sequence of clauses) is an unrestricted LD-derivation. We say that the latter is the unrestricted LD-derivation corresponding to the former. If the mgu’s $\theta_i, \theta_2, \ldots$ of the LD-derivation are relevant then $Q_i \theta_{i+1} \cdots \theta_j = Q_i (\theta_{i+1} \mid Q_i) \cdots (\theta_j \mid Q_i)$ (for any $i < j$). Thus if both derivations are successful then they have the same answer.

Definition A.2 (lift) An LD-derivation $D = Q_0, Q_1, \ldots; \theta_1, \theta_2, \ldots$ is a lift of an unrestricted LD-derivation $E = R_0, R_1, \ldots; \alpha_1, \alpha_2, \ldots; C_1, C_2, \ldots$ when

- $R_0$ is an instance of $Q_0$,
- $D, E$ are of the same length,
- to each $Q_{i-1}$ a variant of a clause $C_i$ has been applied in $D$. 

This notion of lift differs from that of Doets [Doe94, Def. 5.35]. The latter is based on a different notion of derivation, see Sect. 3.1; technically it is a special case of unrestricted derivation. We skip most of technical details of a comparison, we just mention two facts. Let $E$ be as in Definition A.2. If $D$ with relevant mgu’s is a lift of $E$ then the corresponding unrestricted LD-derivation is a lift of $E$ in the sense of Doets. Conversely, if $D' = Q_0, Q_1, \ldots; B_1, B_2, \ldots; C_1, C_2, \ldots$ is a lift of $E$ in the sense of Doets then there exists a lift $D = Q_0, Q_1, \ldots; \theta_1, \theta_2, \ldots$ with relevant mgu’s such that each $Q_i$ is a variant of $Q'_i$, and $Q_0 = Q'_0$. (The proof of Lemma 5.21 of [Doe94] shows, how to obtain $Q_1$ out of $Q'_0 = Q_0, Q'_1, \beta_1, C_1$, by means of an mgu which can be required to be relevant; Lemma 5.24 makes possible iterating this step to produce the whole $D$.)

Our main proof refers to the lifting theorem in the form of [Doe94, Th. 5.37]. The full power of the theorem is not needed here, the following corollary is sufficient.

**Corollary A.3** (lifting) Every unrestricted LD-derivation $E$ starting from a query $R_0$, which is an instance of $Q_0$, has a lift $D$ starting from $Q_0$, with all mgu’s being relevant.

If an LD-derivation $D = Q_0, Q_1, \ldots$ is a lift of an unrestricted LD-derivation $E = R_0, R_1, \ldots$ then each $R_i$ is an instance of $Q_i$. Hence $D$ is successful iff $E$ is successful.

If an LD-derivation $D = Q_0, Q_1, \ldots; \theta_1, \theta_2, \ldots$ (where $\theta_1, \theta_2, \ldots$ are relevant mgu’s) is a lift of an unrestricted LD-derivation $E = R_0, R_1, \ldots; \alpha_1, \alpha_2, \ldots; C_1, C_2, \ldots$ then $R_i, \alpha_i+1, \ldots, \alpha_j$ is an instance of $Q_0, \theta_{i+1}, \ldots, \theta_j$ (for any $i < j$ such that $R_i, \ldots, R_j$ are queries of $D$). In particular, if $E$ is successful then the answer of $E$ is an instance of the answer of $D$.

The corollary follows from the lifting theorem by the two aforementioned properties comparing the lifts of Definition A.2 and those of Doets.

We are ready to begin a proof of Theorem 5.4. It consists of a few lemmas.

**Lemma A.4** Let $P$ be a program and $D = Q_0, \ldots, Q_n; \theta_1, \ldots, \theta_n$ an LD-derivation for $P$. Consider a substitution $\sigma$ and the instances $Q_0, \ldots, Q_n; \theta_1, \ldots, \theta_n$ of $Q_0$ and $Q_n$. Then there exists an unrestricted derivation $D' = Q'_0, \ldots, Q'_n$ for $P$ such that $D$ is a lift of $D'$.

**Proof** The queries of $D'$ are $Q'_i = Q_i; \theta_{i+1}, \ldots, \theta_n$ for $i = 0, \ldots, n$. Let $Q_{i-1} = A, Q_i = (\bar{B}, Q' \theta_i)$, where $H \leftarrow \bar{B}$ is a (variant of a) clause of $P$ and $A \theta_i = H \theta_i$. Then $Q'_i = (A, Q' \theta_i, \ldots, \theta_n; \sigma)$. Applying an instance $(H \leftarrow \bar{B}) \theta_i, \ldots, \theta_n; \sigma$ of the clause, we obtain $(\bar{B}, Q' \theta_i, \ldots, \theta_n; \sigma)$ which is $Q'_i$. $\Box$

**Lemma A.5** Let $S$ be a specification. Let a program $P$ be $c$-correct w.r.t. a call-success specification pre, post, and $Q_0$ be a well-asserted query.

Let $D = Q_0, \ldots, Q_n; \theta_1, \ldots, \theta_n$ be an LD-derivation of $P$, where $Q_0 = A, Q' = (\bar{B}_0, !, \bar{B}_1, Q' \theta_1$ (so the first clause (variant) employed in $D$ is $C = H \leftarrow \bar{B}_0, \bar{B}_1$).

Let $Q_1, \ldots, Q_n$ be a successful subderivation of $D$ for $\bar{B}_0 \theta_1$ (so $Q_n = (!, \bar{B}_1, Q' \theta_1, \ldots, \theta_n$).

Let $S \models Q_0 \sigma_0$ for some ground instance $Q_0 \sigma_0$ of $Q_0$, and $A \sigma_0$ be $c$-covered by $C$ w.r.t. $S$, pre, post.

Then there exists a ground instance $Q_0, \sigma'$ of $Q_0$, such that $S \models Q_0 \sigma'$. Moreover, $Q_0 \sigma_0$ is the first and $Q_0 \sigma'$ is the last query of an unrestricted LD-derivation $D$ such that $D$ is a lift of $D'$.

**Proof** We have $Q_1 = (\bar{B}_0, !, \bar{B}_1, Q' \theta_1)$, and $Q_n = (!, \bar{B}_1, Q' \theta_1, \ldots, \theta_n$. As $P$ is $c$-correct, $\bar{B}_0 \theta_1, \ldots, \theta_n \subseteq \text{post} \cup \{!\}$. Without loss of generality we can assume that $\text{dom}(\sigma_0) = \text{vars}(Q_0)$. Remember that $A \theta_1 = H \theta_1$.

Let $\rho = \theta_1|_{C} = \theta_1|_{H}$. Note that $A \sigma_0$ is an instance of $A$ and of $H$, as $A \sigma_0$ is covered by $C$. Thus $A \sigma_0$ is an instance of $H \theta_1 = H \rho$. As $A \sigma_0$ is $c$-covered by $C$, from Definition 5.2 it follows that $A \sigma_0$ is covered w.r.t. $S$ by $(H \leftarrow \bar{B}_1) \rho$ for any $\eta$ such that $\bar{B}_0 \rho \eta$ is ground, $\bar{B}_0 \rho \eta \subseteq \text{post} \cup \{!\}$ and $\text{dom}(\eta) = \text{vars}(\bar{B}_0 \rho \eta)$.

In particular, this holds for $\eta = (\theta_2, \ldots, \theta_n; \tau)|_{\bar{B}_0 \rho \eta}$ (for any $\tau$ for which $\bar{B}_0 \rho \eta = \bar{B}_0 \theta_1, \ldots, \theta_n \tau$ is ground, and $\text{dom}(\tau) \subseteq \text{vars}(\bar{B}_0 \theta_1, \ldots, \theta_n \tau)$).

Note that $\eta = ((\theta_2, \ldots, \theta_n)|_{\bar{B}_0 \rho \eta}) \tau$. Also $\bar{B}_0 \rho \eta = \bar{B}_0 \theta_1 \rho$, and $(H \leftarrow \bar{B}_1) \rho = (A \leftarrow \bar{B}_1) \theta_1$. Thus $\text{vars}(H \leftarrow \bar{B}_1) \rho \eta \subseteq \text{vars}(A) \cup \text{vars}(\bar{B}_1) \cup \text{vars}(\theta_1)$. By Lemma 3.2 applied to $Q_1$, if a variable $X$ occurs in $(H \leftarrow \bar{B}_1) \rho$ but not in $\bar{B}_0 \rho \eta$ then $X$ does not occur in $\theta_2, \ldots, \theta_n$. Hence $X \theta_2, \ldots, \theta_n = X$, so $X \eta = X \tau = X \theta_2, \ldots, \theta_n \tau$, and thus $(H \leftarrow \bar{B}_1) \rho \eta = (H \leftarrow \bar{B}_1) \theta_1 \tau, \ldots, \theta_n \tau$.

Let $\phi = \theta_1 \ldots, \theta_n$. As $A \sigma_0$ is covered by $(H \leftarrow \bar{B}_1) \phi \tau$, for some ground instance $(H \leftarrow \bar{B}_1) \phi \tau \tau'$ we have $H \phi \tau \tau' = A \phi \tau' = A \sigma_0$ and $B_1 \phi \tau \tau' \subseteq S \cup \{!\}$, where $\text{dom}(\tau') = \text{vars}(H \leftarrow \bar{B}_1) \phi \tau$.
Now (i) $X\sigma_0 = X\varphi t\tau'$ for each variable $X$ occurring in $A$, as $A\sigma_0 = A\varphi t\tau'$. On the other hand, we show that (ii) if $X$ occurs in $Q_0$ but not in $A$ then $X\varphi t\tau' = X$. Take such variable $X$. By Lemma 3.2 applied to $Q_0$, variable $X$ occurs neither in $(\vec{B}_0, \vec{B}_1)\theta_1$, nor in any of the mgu's $\theta_1, \ldots, \theta_n$. Hence $X$ does not occur in $\vec{B}_0\varphi, \vec{B}_1\varphi, A\varphi$. Thus $X\varphi t\tau' = X\varphi t\tau' = X$ (as $X \notin \text{dom}(\varphi)$, $X \notin \text{dom}(\tau) \subseteq \text{vars}(\vec{B}_0\varphi), X \notin \text{dom}(\tau) \subseteq \text{vars}(A \vec{\rightarrow} (\vec{B}_1)\varphi)$). This completes the proof of (ii).

Let us split $\sigma_0$, let $\sigma_1 = \sigma_0|_{\vec{A}}$ and $\sigma_2 = \sigma_0 \setminus \sigma_1$. As $\sigma_0$ is ground, $\sigma_0 = \sigma_1\sigma_2 = \sigma_2\sigma_1$. Now by (i), for any $X \in \text{vars}(A)$ we have $X\sigma_0 = X\varphi t\tau' = X\varphi t\tau'\sigma_2$ (as $X\sigma_0$ is ground). For any $X \in \text{vars}(Q_0)\setminus\text{vars}(A)$ we have $X\sigma_0 = X\varphi t\tau'\sigma_2$; by (ii) it follows $X\sigma_0 = X\varphi t\tau'\sigma_2$. Thus $\sigma_0|_{\vec{A}} = \varphi t\tau'\sigma_2$. Let $\psi = t\tau'\sigma_2$. We have $S \models Q_n\psi$, as $Q_n\psi$ consists of $\psi, \vec{B}_1\varphi\psi$ and of $Q'\varphi\psi$, which is a fragment of $Q_0\sigma_0$; we showed that $B_1\varphi t\tau' \subseteq S \cup \{\}$. Hence $\vec{B}_1\varphi\psi$$\psi \models Q_n\sigma_0$ holds by the premises of the Lemma.

The required instance $Q_n\sigma'$ is $Q_n\psi$. The required unrestricted derivation $D'$ exists by Lemma A.4 (with $Q_0' = Q_0\sigma_0 = Q_0\varphi\psi$ and $Q_n' = Q_n\psi$).

Lemma A.6 Let $T$, $P$, $Q$, $S$, $\pre$, $\post$ be as in Theorem 5.4. Consider a node $Q_0$ of pruned$(T)$ with a ground instance $Q_0\sigma$, such that $Q_0$ is not empty, $S \models Q_0\sigma$, and for $Q_0$ it holds that

\[\text{if the node occurs in a cutting sequence } D \text{ of pruned}(T)\]
\[\text{then it is the introducing node of } D.\] (1)

Then there exists in pruned$(T)$ a descendant $Q_k$ of $Q_0$ satisfying (1), with a ground instance $Q_k\sigma'$ such that $S \models Q_k\sigma'$. Moreover, $Q_0\sigma$, $Q_k\sigma'$ are, respectively, the first and the last query of an unrestricted LD-derivation $D'$ for $P$, w.r.t. $\overset{\text{ars}}{\vec{B}}$.

Proof Outline: We first show that $Q_0$ has a child $Q$ in $T$ such that $S \models \exists Q$. Then we show that $Q$ is a node of pruned$(T)$. If $Q$ does not occur in a cutting sequence of pruned$(T)$ then the required node $Q_k$ is $Q$. Otherwise $Q$ is the second node of a cutting sequence $D_0$ beginning in $Q_0$. In this case the required node $Q_k$ is the child of the last node of $D$. The details follow below.

Let $Q_0 = A, Q'$. Now $A\sigma \in S$, so $A\sigma$ is c-covered, hence covered, w.r.t. $S$ by a clause $C = H \vec{\rightarrow} \vec{B}$ of $P$. Node $Q_0$ has a child $Q = (\vec{B}, Q')\theta$ in $T$ (where $\theta$ is an mgu of $A$ and $H$). Let $A\sigma \vec{\rightarrow} \vec{B}\sigma'$ be a ground instance of $C$ such that the atoms of $\vec{B}\sigma'$ are in $S \cup \{\}$. Let $Q''$ be $\vec{B}\sigma'$. $Q'\sigma$. Obviously, $S \models Q''$, and $Q_0\sigma$, $Q''$ is an unrestricted LD-derivation for $P$. Now LD-derivation $Q_0$, $Q''$ is its lift and, by lifting Corollary A.3, $Q''$ is an instance of $Q$.

Assume that $Q_0$ does not occur in any cutting sequence of pruned$(T)$. Then $Q$ is a node of pruned$(T)$, and $Q$ is the required descendant of $Q_0$. Moreover, derivation $Q_0$, $Q$ is a lift of $Q_0\sigma$, $Q''$.

It remains to consider the case of $Q_0 = A, Q'$ being the introducing node of a cutting sequence $D_0 = Q_0, \ldots, Q_k$ of pruned$(T)$. Note that if in pruned$(T)$ there are two such sequences then one of them is a prefix of the other.9 Such distinct sequences are possible when there are multiple cuts in a clause applied to $Q_0$. Let us assume that $D_0$ is the strongest cutting sequence with the introducing node $Q_0$. Let $\theta_1, \ldots, \theta_j$ be the sequence of mgu’s of $D_0$ viewed as an LD-derivation.

So $Q_k = (\vec{B}_0, \ldots, \vec{B}_1, Q')\theta_1$, where $H \vec{\rightarrow} \vec{B}_0$, $\vec{B}$, $\vec{B}_1$ is a (variant of a) clause of $P$, and $Q_j = (!, \vec{B}_1, Q')\theta_1 \ldots \theta_j$. From c-s-correctness of $P$ w.r.t. pre, post it follows that each selected atom in $D_0$ is in pre, and the answer in $D_0$ for this atom is in post. As post is closed under substitution, each atom of $\vec{B}_0\theta_1 \ldots \theta_j$ is in post $\cup \{\}$. We are ready to show that the node $Q$ is present in pruned$(T)$. Assume it is not. So $Q$ occurs to the right of $D_0$ in $T$, as $Q$ is a child of $Q_0$, and $Q_0$ occurs in pruned$(T)$. Atom $A\sigma$ is c-covered by $C$, which is preceded in $P$ by (a variant of) clause $H \vec{\rightarrow} \vec{B}_0, !, \vec{B}_1$. Let us make explicit the first occurrence of the cut in the clause: let $B_0, !, B_1 = A_0, !, A_1$, where $A_0$ does not contain $A$ and is a prefix of $B_0$.

Consider a ground instance $C' = (A \vec{\rightarrow} \hat{A}_0)\theta_1 \ldots \theta_j \rho$ of $H \vec{\rightarrow} \hat{A}_0$ (the former is an instance of the latter as $A\theta_1 = H\theta_1$). Now $A\sigma$ is an instance of $A \in \pre$, and a ground instance $A\theta_1 \ldots \theta_j \rho$ of $A$ is covered by $H \vec{\rightarrow} \hat{A}_0$ w.r.t. post $\cap \text{HB}$ (as each atom of the body of clause $C'$ is in $(\pre \cap \text{HB} \cup \{\})$). Thus condition 2 of Definition 5.2

9 As if a node $R$ occurs in a cutting sequence of pruned$(T)$ then at most one of its children occurs in a cutting sequence of pruned$(T)$. (Assume that two children do; then one of them is pruned due to the other one).
is violated, and $A \sigma$ is not c-covered by $C$; contradiction. Hence either $Q$ occurs in \textit{pruned}(T) to the left of $Q_1$, or $Q = Q_1$. In the first case (i.e. $Q \neq Q_1$), the node $Q$ is the required descendant of $Q_0$ (with $Q_0$, $Q$ being a lift of $Q \sigma$, $Q''$, as shown above).

In the second case (where $Q = Q_1$), by Lemma A.5, node $Q_j$ has an instance $Q_j \sigma'$ such that $S \models Q_j \sigma'$, there exists an unrestricted LD-derivation $D'_0 = Q_0 \sigma$, ..., $Q_j \sigma'$, and $D_0$ is a lift of $D'_0$. Now the child $Q_k = (\vec{B}_1, Q' \theta_1 \ldots \theta_j)$ of $Q_j$ is the required node of \textit{pruned}(T), $D'$ is $D'_0$, $Q_k \sigma'$, and its lift $D$ is $D_0$, $Q_k$.

\begin{proof}[Proof of Theorem 5.4] Let $Q_0$ be the root of $T$, and $Q_0 \sigma$ be its ground instance such that $S \models Q \sigma$. As no $!$ occur in $Q_0$, $Q_0$ satisfies (1). By induction from Lemma A.6 we obtain that there is a successful or infinite branch $D$ in \textit{pruned}(T), which is a lift of an unrestricted derivation $D'$ beginning with $Q_0 \sigma$. As \textit{pruned}(T) is finite, $D$, $D'$ are successful. Hence the answer $Q_0 \sigma$ of $D'$ is an instance of the answer of $D$.
\end{proof}

We conclude with a proposition which simplifies checking that an atom is c-covered by a clause containing the cut (condition 3 of Definition 5.2).

\begin{proposition} Assume the notation of Definition 5.2. If condition 3 of Definition 5.2 holds for an atom $H \rho \in \text{pre}$ then it holds for any instance $H \rho'$ of $H \rho$ such that $A$ is an instance of $H \rho'$, and $\rho'$ satisfies the requirements of condition 3 (i.e. $\text{dom}(\rho') \subseteq \text{vars}(H)$, $\text{rng}(\rho') \cap \text{vars}(C) \subseteq \text{vars}(H)$, $\text{dom}(\rho') \cap \text{rng}(\rho') = \emptyset$).
\end{proposition}

\begin{lemma} Let $C$ be a clause $H \leftarrow \vec{B}_0 \not\in \vec{B}_1$. Let $S$, $A$, $\rho$, $\eta$, $H \rho$ and $\vec{B}_0 \rho \eta$ be as in condition 3 of Definition 5.2. Let $B_0 = A_1, \ldots, A_{k-1}$ and $B_1 = A_k, \ldots, A_n$. The following conditions (1) and (2) are equivalent.
\begin{enumerate}
  \item $A$ is covered by $(H \leftarrow \vec{B}_1)\rho \eta$ w.r.t. $S$.
  \item There exists a successful LD-derivation for $A$ using in its consecutive steps the clauses $C$, $A_1 \rho \eta$, ..., $A_{k-1} \rho \eta$, then $\neg$ and then some atoms from $S \cup \{!\}$.
\end{enumerate}
\end{lemma}

Note that in (2) all the queries and all the clauses used in the derivation, except $C$, are ground.

\begin{proof} (1) $\Rightarrow$ (2): (1) implies that $A$ is covered by a ground clause $(H \leftarrow \vec{B}_1)\rho \eta$. Construct an LD-derivation $D$ for $A$, using first clause $C \rho \sigma$ and then the clauses as in (2). Its lift is a required derivation.

(2) $\Rightarrow$ (1): Take a derivation as in (2):
\begin{align*}
A \\
(A_1, \ldots, A_n) \theta_1 & \quad \theta_1 \\
\ldots & \quad \ldots \\
(!, A_k, \ldots, A_n) \theta_1 \ldots \theta_k & \quad \theta_k \\
\ldots & \quad \ldots \\
A_n \theta_1 \ldots \theta_{n+1} & \quad \theta_{n+1} \\
\cdots & \quad \cdots \\
\Box & \quad \theta_{n+2}
\end{align*}

Its mgu’s are ground substitutions. We have $A = H \theta_1 = H \theta_1 \ldots \theta_{n+2}$, and the ground clauses used in the derivation are $A_i \theta_1 \ldots \theta_{i+1} = A_i \theta_1 \ldots \theta_{n+2}$ ($i = 1, \ldots, k-1$), and $A_i \theta_1 \ldots \theta_{i+2} = A_i \theta_1 \ldots \theta_{n+2}$ ($i = k, \ldots, n$). Comparing this with (2) gives $A_i \theta_1 \ldots \theta_{n+2} = A_i \rho \eta$, for $i = 1, \ldots, k - 1$, and $A_i \theta_1 \ldots \theta_{n+2} \in S$ for $i = k, \ldots, n$. Hence $\vec{B}_0 \rho \eta = \vec{B}_0 \theta_1 \ldots \theta_{n+2}$. The rest of the proof, roughly speaking, deals with representing substitution $\theta_1 \ldots \theta_{n+2}$ as a composition of $\rho \eta$ and a certain substitution $\sigma$. As a result, we obtain a ground instance of $(H \leftarrow \vec{B}_1)\rho \eta$ which covers $A$.

Now $A = H \rho \delta$ for some ground substitution $\delta$ with $\text{dom}(\delta) = \text{vars}(H\rho)$. So $\theta_1 = (\rho \delta)|_{\text{vars}(H)}$, as $\text{dom}(\theta_1) = \text{vars}(H)$. Note that $\text{dom}(\delta) \cap \text{vars}(C) \subseteq \text{vars}(H)$ (as $\text{dom}(\delta) \subseteq \text{vars}(H) \cup \text{rng}(\rho)$, and from Definition 5.2 we have $\text{rng}(\rho) \cap \text{vars}(C) \subseteq \text{vars}(H)$). Hence $\theta_1 = (\rho \delta)|_{\text{vars}(H)} = (\rho \delta)|_{\text{vars}(C)}$, and thus $C \theta_1 = C \rho \delta$. In particular, $\vec{B}_0 \theta_1 = \vec{B}_0 \rho \delta$. So $\vec{B}_0 \rho \eta = \vec{B}_0 \theta_1 \ldots \theta_{n+2} = \vec{B}_0 \rho \delta \theta_2 \ldots \theta_{n+2}$. Thus $\eta = (\delta \theta_2 \ldots \theta_{n+2})|_{\vec{B}_0 \rho}$ (as $\text{dom}(\eta) = \text{vars}(\vec{B}_0 \rho)$).

Let $\sigma = (\delta \theta_2 \ldots \theta_{n+2}) \setminus \eta$. As $\eta$ and $\sigma$ are ground and with disjoint domains, $\delta \theta_2 \ldots \theta_{n+2} = \eta \cup \sigma = \eta \sigma$. Hence $C \theta_1 \ldots \theta_{n+2} = C \rho \delta \theta_2 \ldots \theta_{n+2} = C \rho \eta \sigma$ (as $C \theta_1 = C \rho \delta$). So $H \rho \eta \sigma = H \theta_1 \ldots \theta_{n+2} = A$ and $A_i \rho \sigma = A_i \theta_1 \ldots \theta_{n+2} \in S$, for $i = k, \ldots, n$. Hence $A$ is covered by $(H \leftarrow A_k, \ldots, A_n)\rho \eta$ w.r.t. $S$. \hfill \square

Proof of Proposition A.7. Let C, the clause used in condition 3, be \( H \leftarrow \overrightarrow{B}_0, !, \overrightarrow{B}_1 \). Let \( \overrightarrow{B}_0 \) be \( A_1, \ldots, A_{k-1} \). We first show that \( \overrightarrow{B}_0 \rho' \) is an instance of \( \overrightarrow{B}_0 \rho \). For some \( \delta \) with \( \text{dom}(\delta) \subseteq \text{vars}(H) \rho \), we have \( H \rho' = H \rho \delta \), so \( \rho' = (\rho \delta)\lvert_H \). Consider a variable \( X \) from \( C \). There are two cases:

1. \( X \in \text{vars}(H) \), thus \( X \rho' = X \rho \delta \).
2. \( X \notin \text{vars}(H) \). So \( X \rho' = X \). Also \( X \notin \text{dom}(\rho) \), as \( \text{dom}(\delta) \subseteq \text{vars}(H) \). From \( \text{dom}(\delta) \subseteq \text{vars}(H) \cup \text{rng}(\rho) \) it follows that \( \text{dom}(\delta) \cap \text{vars}(C) \subseteq \text{vars}(H) \) (as \( \text{rng}(\rho) \cap \text{vars}(C) \subseteq \text{vars}(H) \)). So \( X \notin \text{dom}(\delta) \). Hence \( X \rho \delta = X \) and \( X \rho' = X = X \rho \delta \).

We showed that \( \rho' = (\rho \delta)\lvert_C \). So \( \overrightarrow{B}_0 \rho' = \overrightarrow{B}_0 \rho \delta \). Then each ground instance \( \overrightarrow{B}_0 \rho \eta' \) of \( \overrightarrow{B}_0 \rho \) such that \( \overrightarrow{B}_0 \rho \eta' \in \text{post} \) is an instance of \( \overrightarrow{B}_0 \rho \). (\( \overrightarrow{B}_0 \rho \eta' = \overrightarrow{B}_0 \rho \delta \eta' = \overrightarrow{B}_0 \rho \eta \) where \( \eta = (\delta|_{\overrightarrow{B}_0})\eta' \)).

Assume that condition 3 holds for \( H \rho \). Then for each ground instance \( \overrightarrow{B}_0 \rho \eta' \) as above, where each atom of \( \overrightarrow{B}_0 \rho \eta \) is in \( \text{post} \cup \{!\} \), atom \( A \) is covered w.r.t. \( S \) by \( (H \leftarrow \overrightarrow{B}_1) \rho \eta \). By Lemma A.8 there exists a successful LD-derivation for \( A \) using in its consecutive steps the clauses \( C, A_1 \rho \eta, \ldots, A_{k-1} \rho \eta \), and then some atoms from \( S \cup \{!\} \). As \( A_i \rho \eta = A_i \rho \eta' \) for \( i = 1, \ldots, k-1 \), by Lemma A.8 used in the opposite direction, \( A \) is covered by \( (H \leftarrow \overrightarrow{B}_1) \rho \eta' \).

\[ \square \]

References

[And03] Andrews JH (2003) The witness properties and the semantics of the Prolog cut. TPLP 3(1):1–59
[AP93] Apt KR, Pedreschi D (1993) Reasoning about termination of pure Prolog programs. Inf Compt 106(1):109–157
[Apt97] Apt KR (1997) From logic programming to Prolog. International Series in Computer Science. Prentice Hall Europe, Hemel Hempstead, Hertfordshire
[BC89] Bossi A, Cocconi N (1989) Verifying correctness of logic programs. In: Diaz J, Orejas F (eds) TAPSOFT, Vol.2, Lecture Notes in Computer Science, vol 352. Springer, Berlin Heidelberg, pp 96–110
[Bez93] Bezmi M (1993) Strong termination of logic programs. J Log Program 15(1&2):79–97
[Bil90] Billiaud M (1990) Simple operational and denotational semantics for Prolog with cut. Theor Comput Sci 71(2):193–208
[Cla79] Clark KL (1979) Predicate logic as computational formalism. Technical Report 79/59, Imperial College, London
[Der93] Deransart P (1993) Proof methods of declarative properties of definite programs. Theor Comput Sci 118(2):99–166
[DM88] Drabent W, Małuszyński J (1988) Inductive assertion method for logic programs. Theor Comput Sci 59:133–155
[DM93] Drabent W, Małuszyński J (1993) A Grammatical view of logic programming. The MIT Press, Cambridge, Massachusetts, London, England
[DM05] Drabent W, Milikowska M (2005) Proving correctness and completeness of normal programs – a declarative approach. TPLP 5(6):669–711
[Do94] Doets K (1994) From logic to logic programming. The MIT Press, Cambridge
[Dra15] Drabent W (2015) On completeness of logic programs. In: Logic based program synthesis and transformation, LOPSTR 2014. Revised Selected Papers, Lecture Notes in Computer Science, vol 8981. Springer, Berlin. Extended version in CoRR (2014). arXiv:1411.3015.
[Dra16a] Drabent W (2016) Correctness and completeness of logic programs. ACM Trans Comput Log 17(3):18:1–18:32
[Dra16b] Drabent W (2016) On definite program answers and least Herbrand models. TPLP 16(4):498–508
[dV89] de Vink EP (1989) Comparative semantics for PROLOG with cut. Sci Comput Program 13(1):237–264
[KK14] Kriener J, King A (2014) Semantics for Prolog with cut-revisited. In: Codish M, Sumii E (eds) Functional and Logic Programming—12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4–6, 2014. Proceedings, Lecture Notes in Computer Science, vol 8475. Springer, Berlin, pp 270–284
[Llo87] Lloyd JW (1987) Foundations of logic programming. Second, extended edition. Springer, Berlin
[Mah88] Maher MJ (1988) Equivalences of logic programs. In: Minker J (ed) Foundations of deductive databases and logic programming. Morgan Kaufmann, Los Altos, California, pp 627–658
[NM95] Nilsson U, Małuszyński J (1995) Logic, programming and Prolog (2ed). Wiley, Chichester, West Sussex, England. http://www.ida.liu.se/~ulfml/pp/
[SOS+10] Schneider-Kamp P, Giesl J, Ströder T, Serebrenik A, Thiemann R (2010) Automated termination analysis for logic programs with cut. TPLP 10(4-6):365–381
[Sha83] Shapiro E (1983) Algorithmic program debugging. The MIT Press, Cambridge, Massachusetts, London, England
[Spo00] Spofo F (2000) Operational and goal-independent denotational semantics for Prolog with cut. J Log Program 42(1):1–46
[SS94] Sterling L, Shapiro E (1994) The art of Prolog, 2nd edn. The MIT Press, Cambridge, Massachusetts, London, England

Received 3 July 2015
Accepted in revised form 20 July 2016 by Maurizio Proietti, Hirohisa Seki, and Jim Woodcock
Published online 13 September 2016