On Completeness of Logic Programs

Włodzimierz Drabent(✉)

Institute of Computer Science, Polish Academy of Sciences and IDA,
Linköpings Universitet, Linköping, Sweden
drabent@ipipan.waw.pl

Abstract. Program correctness (in imperative and functional programming) splits in logic programming into correctness and completeness. Completeness means that a program produces all the answers required by its specification. Little work has been devoted to reasoning about completeness. This paper presents a few sufficient conditions for completeness of definite programs. We also study preserving completeness under some cases of pruning of SLD-trees (e.g. due to using the cut).

We treat logic programming as a declarative paradigm, abstracting from any operational semantics as far as possible. We argue that the proposed methods are simple enough to be applied, possibly at an informal level, in practical Prolog programming. We point out importance of approximate specifications.

Keywords: Logic programming · Program completeness · Declarative programming · Approximate specification

1 Introduction

The notion of partial program correctness splits in logic programming into correctness and completeness. Correctness means that all answers of the program are compatible with the specification, completeness – that the program produces all the answers required by the specification.

In this paper we consider definite clause programs, and present a few sufficient conditions for their completeness. We also discuss preserving completeness under pruning of SLD-trees (by e.g. using the cut). We are interested in declarative reasoning, i.e. abstracting from any operational semantics, and treating program clauses as logical formulae. Our goal is simple methods, which may be applied – possibly informally – in actual practical programming.

Related Work. Surprisingly little work was devoted to proving completeness of programs. Hogger [15] defines the notion of completeness, but does not provide any sufficient conditions. Completeness is not discussed in the important monograph [1]. Instead, a characterization is studied of the set of computed instances of an atomic query, in a special case when the set is finite and the answers are ground. In the paper [18] of Kowalski completeness is discussed, but the example proofs concern only correctness. As a sufficient condition for completeness

M. Proietti and H. Seki (Eds.): LOPSTR 2014, LNCS 8981, pp. 261–278, 2015.
DOI: 10.1007/978-3-319-17822-6_15

of a program P he suggests $P \vdash T_S$, where T_S is a specification in a form of a logical theory. The condition seems impractical as it fails when T_S contains auxiliary predicates, not occurring in P. It also requires that all the models of P (including the Herbrand base) are models of the specification. But it seems that such specifications often have a substantially restricted class of models, maybe a single Herbrand model, cf. [6].

Deville [6] provides an approach where correctness and completeness of programs should follow from construction. No direct sufficient criteria for completeness, applicable to arbitrary programs, are given. Also the approach is not declarative, as it is based on an operational semantics of SLDNF-resolution.

Stärk [22] presents an elegant method of reasoning about a broad class of properties of programs with negation, executed under LDNF-resolutions. A tool to verify proofs mechanically was provided. The approach involves a rather complicated induction scheme, so it seems impossible to apply the method informally by programmers. Also, the approach is not fully declarative, as the order of literals in clause bodies is important.

A declarative sufficient condition for program completeness was given by Deransart and Małuszyński [5]. The approach presented here stems from [13], the differences are discussed in [11]. The main contribution since the former version [9,10] is proving completeness of pruned SLD-trees. The author is not aware of any other work on this issue.

This paper, except Sect. 4.2, is an abbreviated version of some parts of [11]. A full version of Sect. 4.2 appeared in [12]. The reader is referred to [11,12] for missing proofs, more examples and further discussion.

Preliminaries. We use the standard notation and definitions [1]. 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 the clauses for p in P.

We assume a fixed alphabet with an infinite set of function symbols. The Herbrand universe will be denoted by $\mathcal{HU}$, the Herbrand base by $\mathcal{HB}$, and the sets of all terms, respectively atoms, by $\mathcal{TU}$ and $\mathcal{TB}$. For an expression (a program) E by $ground(E)$ we mean the set of ground instances of E (ground instances of the clauses of E). $\mathcal{M}_P$ denotes the least Herbrand model of a program P.

By "declarative" (property, reasoning, ...) we mean referring only to logical reading of programs, thus abstracting from any operational semantics. In particular, properties depending on the order of atoms in clauses will not be considered declarative (as they treat equivalent conjunctions differently).

By a computed (respectively correct) answer for a program P and a query Q we mean an instance $Q\theta$ of Q where θ is a computed (correct) answer substitution [1] for Q and P. We often say just *answer* as each computed answer is a correct one, and each correct answer (for Q) is a computed answer (for Q or for some its instance). Thus, by soundness and completeness of SLD-resolution, $Q\theta$ is an answer for P iff $P \models Q\theta$.

Names of variables begin with an upper-case letter. 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 $[a, a|X]$, or $[a, a|a]$, where a is a constant, are not lists). The set of natural numbers will be denoted by $\mathbb{N}$; $f\colon A \hookrightarrow B$ states that f is a partial function from A to B.

The next section introduces the basic notions of specifications, correctness and completeness. Also, advantages of approximate specifications are discussed. The section is concluded with a brief overview on proving correctness. Section 3 discusses proving program completeness. Section 4 deals with proving that completeness is preserved under pruning. We finish with a discussion.

2 Correctness and Completeness

2.1 Specifications

The purpose of a logic program is to compute a relation, or a few relations. A specification should describe these relations. It is convenient to assume that the relations are over the Herbrand universe. To describe such relations, one relation corresponding to each procedure of the program (i.e. to a predicate symbol), it is convenient to use a Herbrand interpretation. Thus a (formal) **specification** is a Herbrand interpretation, i.e. a subset of $\mathcal{HB}$.

2.2 Correctness and Completeness

In imperative and functional programming, correctness usually means that the program results are as specified. In logic programming, due to its non-deterministic nature, we actually have two issues: *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. In terms of specifications and the least Herbrand models we define:

Definition 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. We propose to call a program **fully correct** when it is both correct and complete. If a program P is fully correct w.r.t. a specification S then, obviously, $\mathcal{M}_P = S$.

A program P is correct w.r.t. a specification S iff Q being an answer of P implies $S \models Q$. (Remember that Q is an answer of P iff $P \models Q$.) The program is complete w.r.t. S iff $S \models Q$ implies that Q is an answer of P. (Here our assumption on an infinite set of function symbols is needed [11].)

It is sometimes useful to consider local versions of these notions:

Definition 2. *A* **predicate** *p in P is* **correct** *w.r.t. S when each p-atom of $\mathcal{M}_P$ is in S, and* **complete** *w.r.t. S when each p-atom of S is in $\mathcal{M}_P$.*

An **answer** *Q is* **correct** *w.r.t. S when $S \models Q$.*

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 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 iff it is complete w.r.t. S for any query $A \in S$.

2.3 Approximate Specifications

Often it is difficult, and not necessary, to specify the relations defined by a program exactly; more formally, to require that $\mathcal{M}_P$ is equal to a given specification. Often the relations defined by programs are not exactly those intended by programmers. For instance this concerns the programs in Chap. 3.2 of the textbook [23] defining predicates `member/2`, `append/3`, `sublist/2`, and some others. The defined relations are not those of list membership, concatenation, etc. However this is not an error, as for all intended queries the answers are as for a program defining the intended relations. The exact semantics of the programs is not explained in the textbook; such explanation is not needed. Let us look more closely at `append/3`.

Example 3. 1. The program APPEND

$$app(\,[H|K], L, [H|M]\,) \leftarrow app(\,K, L, M\,). \qquad\qquad app(\,[\,], L, L\,).$$

does not define the relation of list concatenation. For instance, APPEND $\models$ $app([\,], 1, 1)$. In other words, APPEND is not correct w.r.t.

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

where $k * l$ stands for the concatenation of lists k, l. It is however complete w.r.t. S^0_{APPEND}, and correct w.r.t.

$$S_{\text{APPEND}} = \{\, app(k,l,m) \in \mathcal{HB} \mid \text{if } l \text{ or } m \text{ is a list then } app(k,l,m) \in S^0_{\text{APPEND}} \,\}.$$

Correctness w.r.t. S_{APPEND} and completeness w.r.t. S^0_{APPEND} are sufficient to show that APPEND will produce the required results when used to concatenate or split lists. More precisely, the answers for a query $Q = app(s,t,u)$, where t is a list or u is a list, are $app(s\theta, t\theta, u\theta)$, where $s\theta, t\theta, u\theta$ are lists and $s\theta * t\theta = u\theta$. (The lists may be non-ground.)

2. Similarly, the procedures `member/2` and `sublist/2` are complete w.r.t specifications describing the relation of list membership, and the sublist relation. It is easy to provide specifications, w.r.t. which the procedures are correct. For instance, `member/2` is correct w.r.t. $S_{\text{MEMBER}} = \{\, member(t,u) \in \mathcal{HB} \mid$ if $u = [t_1, \ldots, t_n]$ for some $n \geq 0$ then $t = t_i$, for some $0 < i \leq n \,\}$.
3. The exact relations defined by programs are often misunderstood. For instance, in [7, Ex. 15] it is claimed that a program $Prog_1$ defines the relation of list inclusion. In our terms, this means that predicate *included* of $Prog_1$ is correct and complete w.r.t.

$$\left\{\, included(l_1, l_2) \in \mathcal{HB} \;\middle|\; \begin{array}{l} l_1, l_2 \text{ are lists,} \\ \text{every element of } l_1 \text{ belongs to } l_2 \end{array} \right\}.$$

However the correctness does not hold: The program contains a unary clause $included([\,], L)$, so $Prog_1 \models included([\,], t)$ for any term t.

The examples show that in many cases it is unnecessary to know the semantics of a program exactly. Instead it is sufficient to describe it approximately. An **approximate specification** is a pair of specifications S_{compl}, S_{corr}, for completeness and correctness. The intention is that the program is complete w.r.t. the former, and correct w.r.t. the latter: $S_{compl} \subseteq \mathcal{M}_P \subseteq S_{corr}$. In other words, the specifications S_{compl}, S_{corr} describe, respectively, which atoms have to be computed, and which are allowed to be computed. For the atoms from $S_{corr} \setminus S_{compl}$ the semantics of the program is irrelevant. By abuse of terminology, S_{corr} or S_{compl} will sometimes also be called approximate specifications.

2.4 Proving Correctness

We briefly discuss proving correctness, as it is complementary to the main subject of this paper. The approach is due to Clark [4].

Theorem 4 (Correctness). *A sufficient condition for a program P to be correct w.r.t. a specification S is*

> *for each ground instance $H \leftarrow B_1, \ldots, B_n$ of a clause of the program, if $B_1, \ldots, B_n \in S$ then $H \in S$.*

Example 5. Consider a program SPLIT and a specification describing how the sizes of the last two arguments of s are related ($|l|$ denotes the length of a list l):

$$s([\,],[\,],[\,]). \tag{1}$$

$$s([X|Xs],[X|Ys],Zs) \leftarrow s(Xs,Zs,Ys). \tag{2}$$

$$S = \{\, s(l,l_1,l_2) \mid l,l_1,l_2 \text{ are lists, } 0 \leq |l_1| - |l_2| \leq 1 \,\}.$$

SPLIT is correct w.r.t. S, by Theorem 4 (the details are left for the reader, or see [11]). A stronger specification for which SPLIT is correct is shown in Example 11.

The sufficient condition is equivalent to $S \models P$, and to $T_P(S) \subseteq S$.

Notice that the proof method is declarative. The method should be well known, but is often neglected. For instance it is not mentioned in [1], where a more complicated method, moreover not declarative, is advocated. That method is not more powerful than the one of Theorem 4 [13]. See [11,13] for further examples, explanations, references and discussion.

3 Proving Completeness

We first introduce a notion of semi-completeness, and sufficient conditions under which semi-completeness of a program implies its completeness. Then a sufficient condition follows for semi-completeness. We conclude the section with a way of showing completeness directly without employing semi-completeness.

Definition 6. *A* level mapping *is a function* $|\ |\colon \mathcal{HB} \to \mathbb{N}$ *assigning natural numbers to atoms.*

A program P *is* **recurrent** *w.r.t. a level mapping* $|\ |$ *[1,3] if, in every ground instance* $H \leftarrow B_1, \ldots, B_n \in ground(P)$ *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* $H \leftarrow B_1, \ldots, B_n \in ground(P)$ *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.*

The definition of acceptable is more general than that of [1,2] which requires S to be a model of P. Both definitions make the same programs acceptable [11].

Definition 7. *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 P with Q terminates under some selection rule. For a semi-complete program, if a computation for a query Q terminates then all the required by the specification answers for Q have been obtained. Note that a complete program is semi-complete. Also:

Proposition 8 (Completeness). *Let a program* P *be semi-complete w.r.t.* S. *The program is complete w.r.t* S *if*

1. *for each query* $A \in S$ *there exists a finite SLD-tree, or*
   each $A \in S$ *is an instance of a query* Q *for which a finite SLD-tree exists, or*
2. *the program is recurrent, or*
3. *the program is acceptable (w.r.t. a specification* S' *possibly distinct from* S*).*

Proving Semi-completeness. We need the following notion.

Definition 9. *A ground atom* H *is* **covered by a clause** C *w.r.t. a specification* S *[21] 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. *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* $C \in P$.

For instance, given a specification $S = \{p(s^i(0)) \mid i \geq 0\}$, atom $p(s(0))$ is covered both by $p(s(X)) \leftarrow p(X)$ and by $p(X) \leftarrow p(s(X))$.

Now we present a sufficient condition for semi-completeness. Together with Proposition 8 it provides a sufficient condition for completeness.

Theorem 10 (Semi-completeness). *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 11. We show that program SPLIT from Example 5 is complete w.r.t.

$$S_{\text{SPLIT}} = \left\{ \begin{array}{l} s([t_1, \ldots, t_{2n}], [t_1, \cdots, t_{2n-1}], [t_2, \cdots, t_{2n}]), \\ s([t_1, \ldots, t_{2n+1}], [t_1, \cdots, t_{2n+1}], [t_2, \cdots, t_{2n}]) \end{array} \;\middle|\; \begin{array}{l} n \geq 0, \\ t_1, \ldots, t_{2n+1} \in \mathcal{HU} \end{array} \right\},$$

where $[t_k, \cdots, t_l]$ denotes the list $[t_k, t_{k+2}, \ldots, t_l]$, for k, l both odd or both even.

Atom $s([\,],[\,],[\,]) \in S_{\text{SPLIT}}$ is covered by clause (1). For $n > 0$, any atom $A = s([t_1, \ldots, t_{2n}], [t_1, \cdots, t_{2n-1}], [t_2, \cdots, t_{2n}])$ is covered by an instance of (2) with a body $B = s([t_2, \ldots, t_{2n}], [t_2, \cdots, t_{2n}], [t_3, \cdots, t_{2n-1}])$. Similarly, for $n \geq 0$ and any atom $A = s([t_1, \ldots, t_{2n+1}], [t_1, \cdots, t_{2n+1}], [t_2, \cdots, t_{2n}])$, the corresponding body is $B = s([t_2, \ldots, t_{2n+1}], [t_2, \cdots, t_{2n}], [t_3, \cdots, t_{2n+1}])$. In both cases, $B \in S_{\text{SPLIT}}$. (To see this, rename each t_i as t'_{i-1}.) So S_{SPLIT} is covered by SPLIT. Thus SPLIT is semi-complete w.r.t. S_{SPLIT}, by Theorem 10.

Now by Proposition 8 the program is complete, as it is recurrent under the level mapping $|s(t, t_1, t_2)| = |t|$, where $|\,[h|t]\,| = 1 + |t|$ and $|f(t_1, \ldots, t_n)| = 0$ (for any ground terms $h, t, t_1, \ldots, t_n$, and any function symbol f distinct from $[\,|\,]$).

By Theorem 4 the program is also correct w.r.t. S_{SPLIT}, as $S_{\text{SPLIT}} \models \text{SPLIT}$. (The details are left to the reader.) Hence $S_{\text{SPLIT}} = \mathcal{M}_{\text{SPLIT}}$.

Note that the sufficient condition of Theorem 10 is equivalent to $S \subseteq T_P(S)$, which implies $S \subseteq \text{gfp}(T_P)$. It is also equivalent to S being a model of ONLY-IF(P) (see e.g. [8] or [13] for a definition).

The notion of semi-completeness is tailored for finite programs. An SLD-tree for a query Q and an infinite program P may be infinite, but with all branches finite. In such case, if the condition of Theorem 10 holds then P is complete for Q [11].

Proving Completeness Directly. Here we present another, declarative, way of proving completeness; a condition is added to Theorem 10 so that completeness is implied directly. This also works for non-terminating programs. However when termination has to be shown anyway, applying Theorem 10 is usually more convenient.

In this section we allow that a level mapping is a *partial* function $|\,|: \mathcal{HB} \hookrightarrow \mathbb{N}$ assigning natural numbers to some atoms.

Definition 12. *A ground atom H is* **recurrently covered** *by a program P w.r.t. a specification S and a level mapping $|\,|: \mathcal{HB} \hookrightarrow \mathbb{N}$ if H is the head of a ground instance $H \leftarrow B_1, \ldots, B_n$ ($n \geq 0$) of a clause of the program, such that $|H|, |B_1|, \ldots |B_n|$ are defined, $B_1, \ldots, B_n \in S$, and $|H| > |B_i|$ for all $i = 1, \ldots, n$.*

For instance, given a specification $S = \{\, p(s^i(0)) \mid i \geq 0 \,\}$, atom $p(s(0))$ is recurrently covered by a program $\{\, p(s(X)) \leftarrow p(X). \}$ under a level mapping for which $|p(s^i(0))| = i$. No atom is recurrently covered by $\{\, p(X) \leftarrow p(X). \}$. Obviously, if H is recurrently covered by P then it is covered by P. If H is covered by P w.r.t. S and P is recurrent w.r.t. $|\,|$ then H is recurrently covered w.r.t. $S, |\,|$. The same holds for P acceptable w.r.t. an $S' \supseteq S$.

Theorem 13 (Completeness 2). (A reformulation of Theorem 6.1 of [5]). *If, under some level mapping $|\,|: \mathcal{HB} \hookrightarrow \mathbb{N}$, all the atoms from a specification S are recurrently covered by a program P w.r.t. S then P is complete w.r.t. S.*

Example 14. Consider a directed graph E. As a specification for a program describing reachability in E, take $S = S_p \cup S_e$, where

$$S_p = \{\, p(t,u) \mid \text{there is a path from } t \text{ to } u \text{ in } E \,\},$$
$$S_e = \{\, e(t,u) \mid (t,u) \text{ is an edge in } E \,\}.$$

Let P consist of a procedure p: $\{\, p(X,X).\ \ p(X,Z) \leftarrow e(X,Y),\, p(Y,Z).\}$ and a procedure e which is a set of unary clauses describing the edges of the graph. Assume the latter is complete w.r.t. S_e. Notice that when E has cycles then infinite SLD-trees cannot be avoided, and completeness of P cannot be shown by Proposition 8.

To apply Theorem 13, let us define a level mapping for the elements of S such that $|e(t,u)| = 0$ and $|p(t,u)|$ is the length of a shortest path in E from t to u (so $|p(t,t)| = 0$). Consider a $p(t,u) \in S$ where $t \neq u$. Let $t = t_0, t_1, \ldots, t_n = u$ be a shortest path from t to u. Then $e(t,t_1), p(t_1,u) \in S$, $|p(t,u)| = n$, $|e(t,t_1)| = 0$, and $|p(t_1,u)| = n-1$. Thus $p(t,u)$ is recurrently covered by P w.r.t. S and $|\ |$. The same trivially holds for the remaining atoms of S. So P is complete w.r.t. S.

4 Pruning SLD-Trees and Completeness

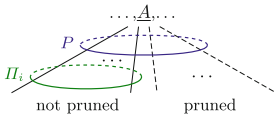
Pruning some parts of SLD-trees is often used to improve efficiency of programs. It is implemented by using the cut, the if-then-else construct of Prolog, or built-ins, like `once/1`. Pruning preserves the correctness of a logic program, it also preserves termination under a given selection rule, but may violate the program's completeness. We now discuss proving that completeness is preserved.

By a **pruned SLD-tree** for a program P and a query Q we mean a tree with the root Q which is a connected subgraph of an SLD-tree for P and Q. By an *answer* of a pruned SLD-tree we mean the computed answer of a successful SLD-derivation which is a branch of the tree. We will say that a pruned SLD-tree T with root Q is **complete** w.r.t. a specification S if, for any ground $Q\theta$, $S \models Q\theta$ implies that $Q\theta$ is an instance of an answer of T. Informally, such a tree produces all the answers for Q required by S.

We present two approaches for proving completeness of pruned SLD-trees. The first one is based on viewing pruning as skipping certain clauses while building the children of a node. The other deals with a restricted usage of the cut.

4.1 Pruning as Clause Selection

To facilitate reasoning about the answers of pruned SLD-trees, we will now view pruning as applying only certain clauses while constructing the children of a node. So we introduce subsets $\Pi_1, \ldots, \Pi_n$ of P. The intention is that for each node the clauses of one Π_i are used. Programs $\Pi_1, \ldots, \Pi_n$ may be not disjoint.



Definition 15. *Given programs $\Pi_1, \ldots, \Pi_n$ $(n > 0)$, a* **c-selection rule** *is a function assigning to a query Q' an atom A in Q' and one of the programs $\emptyset, \Pi_1, \ldots, \Pi_n$.*

A **csSLD-tree** *(cs for clause selection) for a query Q and programs $\Pi_1, \ldots, \Pi_n$, via a c-selection rule R, is constructed as an SLD-tree, but for each node its children are constructed using the program selected by the c-selection rule. An answer of a csSLD-tree is defined in the expected way.*

A c-selection rule may choose the empty program, thus making a given node a leaf. Notice that a csSLD-tree for Q and $\Pi_1, \ldots, \Pi_n$ is a pruned SLD-tree for Q and $\bigcup_i \Pi_i$. Conversely, for each pruned SLD-tree T for Q and a (finite) program P there exist $n > 0$, and $\Pi_1, \ldots, \Pi_n \subseteq P$ such that T is a csSLD-tree for Q and $\Pi_1, \ldots, \Pi_n$.

Describing pruning by a c-selection rule is, in a sense, abstract. It does not refer directly to the control constructs in the program. The correspondence between the program and the c-selection rule may be not obvious [11]. A single cut, or if-then-else, may prune children of many nodes in a tree, modifying the behaviour of many procedures of the program. However Examples 19, 20, 21 below suggest that in many cases this difficulty is not substantial.

Example 16. We show that completeness of each of $\Pi_1, \ldots, \Pi_n$ is not sufficient for completeness of a csSLD-tree for $\Pi_1, \ldots, \Pi_n$. Consider a program P:

$$q(X) \leftarrow p(Y, X). \tag{3}$$
$$p(Y, 0). \tag{4}$$
$$p(a, s(X)) \leftarrow p(a, X). \tag{5}$$
$$p(b, s(X)) \leftarrow p(b, X). \tag{6}$$

and programs $\Pi_1 = \{(3), (4), (6)\}$, $\Pi_2 = \{(3), (4), (5)\}$, As a specification for completeness consider $S_0 = \{\, q(s^j(0)) \mid j \geq 0 \,\}$. Each of the programs Π_1, Π_2, P is complete w.r.t. S_0. Assume a c-selection rule R choosing alternatively Π_1, Π_2 along each branch of a tree. Then the csSLD-tree for $q(s^j(0)) \in S_0$ via R (where $j > 2$) has no answers, thus the tree is not complete w.r.t. S_0.

Consider programs $P, \Pi_1, \ldots, \Pi_n$ and specifications $S, S_1, \ldots, S_n$, such that $P \supseteq \bigcup_{i=1}^n \Pi_i$ and $S = \bigcup_{i=1}^n S_i$. The intention is that each S_i describes which answers are to be produced by using Π_i in the first resolution step. We will call $\Pi_1, \ldots, \Pi_n, S_1, \ldots, S_n$ a **split** (of P and S). Note that $\Pi_1, \ldots, \Pi_n$ or $S_1, \ldots, S_n$ may be not disjoint.

Definition 17. *Let $\mathcal{S} = \Pi_1, \ldots, \Pi_n, S_1, \ldots, S_n$ be a split, and $S = \bigcup S_i$.*

Specification S_i is **suitable** *for an atom A w.r.t. $\mathcal{S}$ when no instance of A is in $S \setminus S_i$. (In other words, when ground$(A) \cap S \subseteq S_i$.) We also say that a program Π_i is* **suitable** *for A w.r.t. $\mathcal{S}$ when S_i is.*

A c-selection rule is **compatible** *with $\mathcal{S}$ if for each non-empty query Q it selects an atom A and a program Π, such that*

– *$\Pi \in \{\Pi_1, \ldots, \Pi_n\}$ is suitable for A w.r.t. $\mathcal{S}$, or*
– *none of $\Pi_1, \ldots, \Pi_n$ is suitable for A w.r.t. $\mathcal{S}$ and $\Pi = \emptyset$ (so Q is a leaf).*

A csSLD-tree for $\Pi_1, \ldots, \Pi_n$ via a c-selection rule compatible with $\mathcal{S}$ is said to be **weakly compatible** *with $\mathcal{S}$. The tree is* **compatible** *with $\mathcal{S}$ iff for each its nonempty node some Π_i is selected.*

The intuition is that when Π_i is suitable for A then S_i is a fragment of S sufficient to deal with A. It describes all the answers for query A required by S.

The reason of incompleteness of the trees in Example 16 may be understood as selecting a Π_i not suitable for the selected atom. Take $\mathcal{S} = \Pi_1, \Pi_2, S_0 \cup S'_1, S_0 \cup S'_2$, where $S'_1 = \{\, p(b, s^i(0)) \mid i \geq 0 \,\}$ and $S'_2 = \{\, p(a, s^i(0)) \mid i \geq 0 \,\}$. In the incomplete trees, Π_1 is selected for an atom $A = p(a, u)$, or Π_2 is selected for an atom $B = p(b, u)$ (where $u \in \mathcal{TU}$). However Π_1 is not suitable for A whenever A has an instance in S (as then $ground(A) \cap S \not\subseteq S_0 \cup S'_1$); similarly for Π_2 and B.

When Π_i is suitable for A then if each atom of S_i is covered by Π_i (w.r.t. S) then using for A only the clauses of Π_i does not impair completeness w.r.t. S:

Theorem 18. *Let $P \supseteq \bigcup_{i=1}^{n} \Pi_i$ (where $n > 0$) be a program, $S = \bigcup_{i=1}^{n} S_i$ a specification, and T a csSLD-tree for $\Pi_1, \ldots, \Pi_n$. If*

1. *for each $i = 1, \ldots, n$, all the atoms from S_i are covered by Π_i w.r.t. S, and*
2. *T is compatible with $\Pi_1, \ldots, \Pi_n, S_1, \ldots, S_n$,*
3. *(a) T is finite, or*
   (b) program P is recurrent, or
   (c) P is acceptable (possibly w.r.t. a specification distinct from S), and T is built under the Prolog selection rule

then T is complete w.r.t. S.

We now show three examples of applying this theorem.

Example 19. The following program SAT0 is a simplification of a fragment of the SAT solver of [16] discussed in [9]. Pruning is crucial for the efficiency and usability of the original program.

$$p(P{-}P,\, [\,]). \tag{7}$$
$$p(V{-}P,\, [B|T]) \leftarrow q(V{-}P,\, [B|T]). \tag{8}$$
$$p(V{-}P,\, [B|T]) \leftarrow q(B,\, [V{-}P|T]). \tag{9}$$
$$q(V{-}P,\, \_) \leftarrow V = P. \tag{10}$$
$$q(\,\_,\, [A|T]) \leftarrow p(A, T). \tag{11}$$
$$P = P. \tag{12}$$

The program is complete w.r.t. a specification

$$S = \left\{ \begin{array}{l} p(t_0{-}u_0, [t_1{-}u_1, \ldots, t_n{-}u_n]), \\ q(t_0{-}u_0, [t_1{-}u_1, \ldots, t_n{-}u_n]) \end{array} \;\middle|\; \begin{array}{l} n \geq 0,\ t_0, \ldots, t_n, u_0, \ldots, u_n \in \mathbb{T}, \\ t_i = u_i \text{ for some } i \in \{0, \ldots, n\} \end{array} \right\} \cup S_{=}$$

where $\mathbb{T} = \{false, true\} \subseteq \mathcal{HU}$, and $S_{=} = \{\, t{=}t \mid t \in \mathcal{HU} \,\}$. We omit a completeness proof, mentioning only that SAT0 is recurrent w.r.t. a level mapping $|p(t,u)| = 2|u|{+}2$, $|q(t,u)| = 2|u|{+}1$, $|{=}(t,u)| = 0$, where $|u|$ is as in Example 11.

The first case of pruning is due to redundancy within (8), (9); both $\Pi_1 = \text{SAT0} \setminus \{(9)\}$ and $\Pi_2 = \text{SAT0} \setminus \{(8)\}$ are complete w.r.t. S. For any selected atom at most one of (8), (9) is to be used, and the choice is dynamic. As the following reasoning is independent from this choice, we omit further explanations.

So in such pruned SLD-trees the children of each node are constructed using one of programs Π_1, Π_2. Thus they are csSLD-trees for Π_1, Π_2. They are compatible with $\mathcal{S} = \Pi_1, \Pi_2, S, S$ (as Π_1, Π_2 are trivially suitable for any A, due to $S_i = S$ and $S \setminus S_i = \emptyset$ in Definition 17). Each atom of S is covered w.r.t. S both by Π_1 and Π_2. As SAT0 is recurrent, by Theorem 18, each such tree is complete w.r.t. S.

Example 20. We continue with program SAT0 and specification S from the previous example, and add a second case of pruning. When the selected atom is of the form $A = q(s_1, s_2)$ with a ground s_1 then only one of clauses (10), (11) is needed – (10) when s_1 is of the form $t{-}t$, and (11) otherwise. The other clause can be abandoned without losing the completeness w.r.t. S.[1]

Actually, SAT0 is included in a bigger program, say $P = \text{SAT0} \cup \Pi_0$. We skip the details of Π_0, let us only state that P is recurrent, Π_0 does not contain any clause for p or for q, and that P is complete w.r.t. a specification $S' = S \cup S_0$ where S_0 does not contain any p- or q-atom. (Hence each atom of S_0 is covered by Π_0 w.r.t. S'.)

To formally describe the trees for P resulting from both cases of pruning, consider $\mathcal{S} = \Pi_0, \ldots, \Pi_5, S_0, \ldots, S_5$, where

$$\begin{array}{llll}
\Pi_1 = \{(7),(8)\}, & \Pi_2 = \{(7),(9)\}, & S_1 = S_2 = S \cap \{\, p(s,u) \mid s,u \in \mathcal{HU} \,\}, \\
\Pi_3 = \{(10)\}, & & S_3 = S \cap \{\, q(t{-}t,s) \mid t,s \in \mathcal{HU} \,\}, \\
\Pi_4 = \{(11)\}, & & S_4 = S \cap \{\, q(t{-}u,s) \mid t,u,s \in \mathcal{HU}, t \neq u \,\}, \\
\Pi_5 = \{(12)\}, & & S_5 = S_{=}.
\end{array}$$

Each atom from S_i is covered by Π_i w.r.t. S' (for $i = 0, \ldots, 5$). For each q-atom with its first argument ground, Π_3 or Π_4 (or both) is suitable. For each remaining atom from $\mathcal{TB}$, a program from $\Pi_0, \Pi_1, \Pi_2, \Pi_5$ is suitable.

Consider a pruned SLD-tree T for P (employing the two cases of pruning described above). Assume that each q-atom selected in T has its first argument ground. Then T is a csSLD-tree compatible with $\mathcal{S}$. From Theorem 18 it follows that T is complete w.r.t. S'.

The restriction on the selected q-atoms is implemented by means of Prolog delays. This is done in such a way that, for the intended initial queries, floundering is avoided [16] (i.e. an atom is selected in each query). So the obtained

[1] The same holds for A of the form $q(s_{11}{-}s_{11}, s_2)$, or $q(s_{11}{-}s_{12}, s_2)$ with non-unifiable s_{11}, s_{12}. The pruning is implemented using the if-then-else construct in Prolog: `q(V-P,[A|T]) :- V = P -> true; p(A,T).` (And the first case of pruning by `p(V-P,[B|T]) :- nonvar(V) -> q(V-P,[B|T]); q(B,[V-P|T]).`)

pruned trees are as T above, and the pruning preserves completeness of the program.

Example 21. A Prolog program $\{nop(adam, 0) \leftarrow !.\ nop(eve, 0) \leftarrow !.\ nop(X, 2).\}$ is an example of difficulties and dangers of using the cut in Prolog. Due to the cut, for an atomic query A only the first clause with the head unifiable with A will be used. The program can be seen as logic program $P = \Pi_1 \cup \Pi_2 \cup \Pi_3$ executed with pruning, where (for $i = 1, 2, 3$) Π_i is the i-th clause of the program with the cut removed. The intended meaning is $S = S_1 \cup S_2 \cup S_3$, where $S_1 = \{nop(adam, 0)\}$, $S_2 = \{nop(eve, 0)\}$, and $S_3 = \{nop(t, 2) \in \mathcal{HB} \mid t \notin \{adam, eve\}\}$. Note that all the atoms from S_i are covered by Π_i (for $i = 1, 2, 3$). (We do not discuss here the (in)correctness of the program, but see [11].)

Let $\mathcal{S}$ be $\Pi_1, \Pi_2, \Pi_3, S_1, S_2, S_3$. Consider a query $A = nop(t, Y)$ with a ground t. If $t = adam$ then $ground(A) \cap S = S_1$, and only Π_1 is suitable for A w.r.t. $\mathcal{S}$, if $t = eve$ then only Π_2 is. For $t \notin \{adam, eve\}$ the suitable program is Π_3. So for the query A the pruning due to the cuts results in selecting a suitable Π_i, and the obtained csSLD-tree is compatible with $\mathcal{S}$. By Theorem 18 the tree is complete w.r.t. S.

For a query $nop(X, Y)$ or $nop(X, 0)$ only the first clause, i.e. Π_1, is used. However Π_1 is not suitable for the query (w.r.t. $\mathcal{S}$), and the csSLD-tree is not compatible with $\mathcal{S}$. The tree is not complete (w.r.t. S).

4.2 The Cut in the Last Clause

The previous approach is based on a somehow abstract semantics in which pruning is viewed as clause selection. Now we present an approach referring directly to Prolog with the cut. However the usage of the cut is restricted to the last clause of a procedure. The author expects that the general case could be conveniently studied in the context of programs with negation (because if $H \leftarrow A_1, !, A_2$ is followed by $H \leftarrow A_3$ then the latter clause is used only if A_1 fails). We consider LD-resolution, as interaction of the cut with delays introduces additional complications.

We need to reason about the atoms selected in the derivations. So we employ a (non-declarative) approach to reason about LD-derivations, presented in [1]. A specification in this approach, let us call it **call-success specification**, is a pair $pre, post \in \mathcal{TB}$ of sets of atoms, closed under substitution. A program is correct w.r.t. such specification, let us say **c-s-correct**, when in each LD-derivation every selected atom is in pre and each corresponding computed answer is in $post$, provided that the derivation begins with an atomic query from pre. The same holds for non-atomic initial queries, provided that they satisfy a certain condition (are *well asserted* [1]). See [1] or [13] for further explanations, and for a sufficient criterion for c-s-correctness (programs satisfying it are called *well asserted*).

By $vars(E)$ we denote the set of variables occurring in an expression E. For a substitution $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$, let $dom(\theta) = \{X_1, \ldots, X_n\}$, and $rng(\theta) = vars(\{t_1, \ldots, t_n\})$. We begin with generalizing the notion of an atom covered by a clause.

Definition 22. Let S be a specification, and $pre, post$ a call-success specification. A ground atom A is **adjustably covered** by a clause C w.r.t. S and $pre, post$ if A is covered by C and the cut does not occur in C, or the following three conditions hold:

1. C is $H \leftarrow A_1, \ldots, A_{k-1}, !, A_k, \ldots, A_n$,
2. A is covered by $H \leftarrow A_1, \ldots, A_{k-1}$ w.r.t. S,
3. – for any instance $H\rho \in pre$ such that A is an instance of $H\rho$,
   – for any ground instance $(A_1, \ldots, A_{k-1})\rho\eta$ such that $A_1\rho\eta, \ldots, A_{k-1}\rho\eta \in post$,
   – A is covered by $(H \leftarrow A_k, \ldots, A_n)\rho\eta$ w.r.t. S,

   where $dom(\rho) \subseteq vars(H)$, $rng(\rho) \cap vars(C) \subseteq vars(H)$, $dom(\rho) \cap rng(\rho) = \emptyset$, and $dom(\eta) = vars((A_1, \ldots, A_{k-1})\rho)$.

Informally, condition 3 says that A could be produced out of each "related" answer for $A_1, \ldots, A_{k-1}$, and some answer for $A_k, \ldots, A_n$ specified by S. Note that if A is adjustably covered by C w.r.t. S, $pre, post$, where $S \subseteq post$, then A is covered by C w.r.t. S.

Now we are ready to present the sufficient condition for completeness.

Theorem 23 ([12]). *Let S be a specification, $pre, post$ a call-success specification, where $S \subseteq post$. Let T be a pruned LD-tree for a program P and an atomic query Q, where pruning is due to the cut occurring in the last clause(s) of some procedure(s) of P. If*

- *T is finite, $Q \in pre$, P is c-s-correct w.r.t. $pre, post$, and*
- *each $A \in S$ is adjustably covered by a clause of P w.r.t. S and $pre, post$*

then T is complete w.r.t. S.

For a non-atomic initial query Q, the condition $Q \in pre$ should be replaced by Q is well asserted w.r.t. $pre, post$ (see [1] for a definition).

We now show two examples of applying this theorem to proving completeness of pruned trees.

Example 24. Consider a program IN:

$$in([\,], L). \qquad m(E, [E|L]).$$
$$in([H|T], L) \leftarrow m(H, L), !, in(T, L). \qquad m(E, [H|L]) \leftarrow m(E, L).$$

and specifications

$S = S_m \cup S_{in}$, $pre = pre_m \cup pre_{in}$, $post = post_m \cup post_{in}$, where
$pre_m = \{\, m(u,t) \in \mathcal{TB} \mid t \text{ is a list} \,\}$,
$pre_{in} = \{\, in(u,t) \in \mathcal{HB} \mid u, t \text{ are ground lists} \,\}$,
$post_m = \{\, m(t_i, [t_1, \ldots, t_n]) \in \mathcal{TB} \mid 1 \le i \le n \,\}$,
$post_{in} = \{\, in([u_1, \ldots, u_m], [t_1, \ldots, t_n]) \in \mathcal{HB} \mid \{u_1, \ldots, u_m\} \subseteq \{t_1, \ldots, t_n\} \,\}$,
$S_m = post_m \cap \mathcal{HB}$, $S_{in} = post_{in}$.

The program is c-s-correct w.r.t. $pre, post$ (we skip a proof). We show that each atom $A = in(u,t) \in S_{in}$, where $u = [u_1, \ldots, u_m]$, $m > 0$, is adjustably covered by the second clause C of IN. Let C_0 be $in([H|T], L) \leftarrow m(H, L)$. Now A is covered by C_0 w.r.t. S ($A \leftarrow m(u_1, t)$ is a relevant ground instance of C_0).

Take an instance $in([H|T], L)\rho \in pre$ of the head of C. The instance is ground, and the whole $C\rho$ is ground. So in Definition 22, $\rho\eta = \rho$. If A is an instance of (thus equal to) $in([H|T], L)\rho$ then $in(T, L)\rho = in([u_2, \ldots, u_m], t) \in S$ (as $A \in S$). Thus A is covered by $(in([H|T], L) \leftarrow in(T, L))\rho$.

Thus A is adjustably covered by C. It is easy to check that all the remaining atoms of S are covered by IN w.r.t. S, and that IN is recurrent (for $|m(s,t)| = |t|$, $|in(s,t)| = |s| + |t|$, $|t|$ as in Example 11). Thus each LD-tree for IN and a query $Q \in pre$ is finite. By Theorem 23, each such tree pruned due to the cut is complete w.r.t. S. Notice that condition 3 does not hold when non ground arguments of in are allowed in pre_{in}, and that for such queries some answers may be pruned.

Before the next example we introduce a property, which simplifies checking that an atom is adjustably covered by a clause with the cut.

Lemma 25 ([12])**.** If condition 3 of Definition 22 holds for an atom $H\rho \in pre$ then it holds for any its instance $H\rho'$ such that A is an instance of $H\rho'$, and ρ' satisfies the requirements of condition 3 (i.e. $dom(\rho') \subseteq vars(H)$, $rng(\rho') \cap vars(C) \subseteq vars(H)$, $dom(\rho') \cap rng(\rho') = \emptyset$).

Example 26. Consider a program P:

$$p(X,Z) \leftarrow q(X,Y), !, r(Y,Z). \qquad \begin{array}{ll} q(a,a) & r(a,c) \\ q(a,a') & r(a',c) \\ q(b,b) & \end{array}$$

and specifications

$$\begin{array}{l} S = \{\, p(a,c), q(a,a'), q(b,b), r(a,c), r(a',c) \,\}, \\ post = S \cup \{q(a,a)\}, \\ pre = \{\, p(a,t) \mid t \in \mathcal{TU} \,\} \cup \{\, q(a,t) \mid t \in \mathcal{TU} \,\} \cup \{\, r(t,u) \mid t,u \in \mathcal{TU} \,\} \end{array}$$

The program is c-s-correct w.r.t. $pre, post$ (we skip a proof). To check that atom $p(a,c) \in S$ is adjustably covered by the first clause of P, note first that it is covered w.r.t. S by $p(a,c) \leftarrow q(a,a')$. By Lemma 25, it is sufficient to check condition 3 of Definition 22 for $\rho = \{X/a\}$, as $p(X,Z)\rho = p(a,Z)$ is a most general p-atom in pre. If $q(X,Y)\rho\eta \in post$ 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, $p(a,c) \leftarrow r(Y\eta, c)$ is a ground instance of $(p(X,Z) \leftarrow r(Y,Z))\rho\eta$ (i.e. of $p(a,Z) \leftarrow r(Y\eta, Z)$) covering $p(a,c)$ w.r.t. S.

The remaining atoms of S are trivially covered by the unary clauses of P. The LD-tree for P and $Q = p(a,Z)$ is finite, hence the LD-tree pruned due to the cut is complete w.r.t. S by Theorem 23.

5 Discussion

Declarativeness. Without declarative ways of reasoning about correctness and completeness of programs, logic programming would not deserve to be called a declarative programming paradigm. The sufficient condition for proving correctness (Theorem 4), that for semi-completeness of Theorem 10, and those for completeness of Proposition 8(2) and Theorem 13 are declarative. Also, the sufficient condition for completeness of pruned trees (Theorem 18), based on clause selection, to a substantial extent abstracts from the operational semantics. On the other hand, the sufficient conditions for program completeness of Propositions 8(1) and 8(3) are not declarative, as they refer to program termination, or depend on the order of atoms in clause bodies.

Declarative completeness proofs employing Proposition 8(2) or Theorem 13 imply termination, or require reasoning similar to that in termination proofs. So proving completeness by means of semi-completeness and termination may be a reasonable compromise between declarative and non-declarative reasoning, as termination has to be shown anyway in most of practical cases.

Granularity of Proofs. Note that the sufficient condition for correctness deals with single clauses, that for semi-completeness – with procedures, and those for completeness take into account a whole program.

Incompleteness Diagnosis. There is a close relation between completeness proving and incompleteness diagnosis [21]. As the reason of incompleteness, a diagnosis algorithm finds an atom from S that is not covered by the program. Thus it finds a reason for violating the sufficient conditions for semi-completeness and completeness of Theorems 10 and 13. So what is diagnosed is lack of semi-completeness. (As the algorithm works with a finite SLD-tree for a program P and a query Q, incompleteness of P for Q implies that P is not semi-complete.)

Approximate Specifications. We found that approximate specifications are crucial in avoiding unnecessary complications in dealing with correctness and completeness of programs (cf. Sect. 2.3, [9,11,13]). For instance, in the main example of [9] (and in its simpler version in Examples 19, and 20) finding an exact specification is not easy, and is unnecessary. The required property of the program is described more conveniently by an approximate specification. Moreover, as this example shows, in program development the semantics of (common predicates in) the consecutive versions of a program may differ. What is unchanged is correctness and completeness w.r.t. an approximate specification.

Approximate Specifications in Program Development. This suggests a generalization of the paradigm of program development by semantics preserving program transformations [19,20]: it is useful and natural to use transformations which only preserve correctness and completeness w.r.t. an approximate specification.

Approximate Specifications in Debugging. In declarative diagnosis [21] the programmer is required to know the exact intended semantics of the program. This is a substantial obstacle to using declarative diagnosis in practice. Instead, an approximate specification can be used, with the specification for correctness (respectively completeness) applied in incorrectness (incompleteness) diagnosis. See [11] for discussion and references.

Interpretations as Specifications. This work uses specifications which are interpretations. (The same kind of specifications is used, among others, in [1], and in declarative diagnosis.) There are however properties which cannot be expressed by such specifications [13]. For instance one cannot express that some instance of an atomic query A should be an answer; one has to specify the actual instance(s). Other approach is needed for such properties, possibly with specifications which are logical theories (where axioms like $\exists X.\, A$ can be used).

Applications. We want to stress the simplicity and naturalness of the sufficient conditions for correctness (Theorem 4) and semi-completeness (Theorem 10, the condition is a part of each discussed sufficient condition for completeness). Informally, the first one says that the clauses of a program should produce only correct conclusions, given correct premises. The other says that each ground atom that should be produced by P can be produced by a clause of P out of atoms produced by P. The author believes that this is a way a competent programmer reasons about (the declarative semantics of) a logic program.

Paper [9] illustrates practical applicability of the methods presented here. It shows a systematic construction of a non-trivial Prolog program (the SAT solver of [16]). Starting from a formal specification, a definite clause logic program is constructed hand in hand with proofs of its correctness, completeness, and termination under any selection rule. The final Prolog program is obtained by adding control to the logic program (delays and pruning SLD-trees). Adding control preserves correctness and termination. However completeness may be violated by pruning, and by floundering related to delays. By Theorem 18, the program with pruning remains complete.[2] Proving non-floundering is outside of the scope of this work. See [14] for a related analysis algorithm, applicable in this case [17].

The example shows how well "logic" could be separated from "control." The whole reasoning related to correctness and completeness can be done declaratively, abstracting from any operational semantics.

Future Work. A natural continuation is developing completeness proof methods for programs with negation (a first step was made in [13]), maybe also for constraint logic programming and CHR (constraint handling rules). Further examples of proofs are necessary. An interesting task is formalizing and automatizing the proofs, a first step is formalization of specifications. Another issue is overcoming the limitation described in *Interpretations as specifications* above.

[2] In [9] a weaker version of Theorem 18 was used, and one case of pruning was discussed informally. A proof covering both cases of pruning is illustrated here in Example 20.

Conclusion. Reasoning about completeness of logic program has been, surprisingly, almost neglected. This paper presents a few sufficient conditions for completeness. As an intermediate step we introduced a notion of semi-completeness. The presented methods are, to a large extent, declarative. Examples suggest that the approach is applicable – maybe at informal level – in practice of Prolog programming. The approach is augmented by two methods of proving completeness in presence of pruning.

References

1. Apt, K.R.: From Logic Programming to Prolog. International Series in Computer Science. Prentice-Hall, Upper Saddle River (1997)
2. Apt, K.R., Pedreschi, D.: Reasoning about termination of pure Prolog programs. Inf. Comput. **106**(1), 109–157 (1993)
3. Bezem, M.: Strong termination of logic programs. J. Log. Program. **15**(1&2), 79–97 (1993)
4. Clark, K.L.: Predicate logic as computational formalism. Technical report 79/59, Imperial College, London (1979)
5. Deransart, P., Małuszyński, J.: A Grammatical View of Logic Programming. The MIT Press, Cambridge (1993)
6. Deville, Y.: Logic Programming: Systematic Program Development. Addison-Wesley, Reading (1990)
7. Deville, Y., Lau, K.-K.: Logic program synthesis. J. Log. Program. **19**(20), 321–350 (1994)
8. Doets, K.: From Logic to Logic Programming. The MIT Press, Cambridge (1994)
9. Drabent, W.: Logic + control: an example. In: Dovier, A., Santos Costa, V. (eds.) Technical Communications of ICLP 2012. LIPIcs, vol. 17, pp. 301–311 (2012). http://drops.dagstuhl.de/opus/volltexte/2012/3631
10. Drabent, W.: Logic + control: an example of program construction, CoRR. abs/1110.4978 (2012). http://arxiv.org/abs/1110.4978.
11. Drabent, W.: Correctness and completeness of logic programs, CoRR. abs/1412.8739 (2014). http://arxiv.org/abs/1412.8739
12. Drabent, W.: On completeness of logic programs, CoRR. abs/1411.3015(2014). http://arxiv.org/abs/1411.3015
13. Drabent, W., Miłkowska, M.: Proving correctness and completeness of normal programs - a declarative approach. Theory Pract. Log. Program. **5**(6), 669–711 (2005)
14. Genaim, S., King, A.: Inferring non-suspension conditions for logic programs with dynamic scheduling. ACM Trans. Comput. Log. **9**(3), 17:1–17:43 (2008)
15. Hogger, C.J.: Introduction to Logic Programming. Academic Press, London (1984)
16. Howe, J.M., King, A.: A pearl on SAT and SMT solving in Prolog. Theor. Comput. Sci. **435**, 43–55 (2012)
17. King, A.: Personal communication, March 2012
18. Kowalski, R.A.: The relation between logic programming and logic specification. In: Hoare, C., Shepherdson, J. (eds.) Mathematical Logic and Programming Languages, pp. 11–27. Prentice-Hall, Upper Saddle River (1985). Also in Phil. Trans. R. Soc. Lond. A, **312**, 345–361(1984)
19. Pettorossi, A., Proietti, M.: Transformation of logic programs: foundations and techniques. J. Log. Program. **19/20**, 261–320 (1994)

20. Pettorossi, A., Proietti, M., Senni, V.: The transformational approach to program development. In: Dovier, A., Pontelli, E. (eds.) GULP. LNCS, vol. 6125, pp. 112–135. Springer, Heidelberg (2010)
21. Shapiro, E.: Algorithmic Program Debugging. The MIT Press, Cambridge (1983)
22. Stärk, R.F.: The theoretical foundations of LPTP (a logic program theorem prover). J. Log. Program. **36**(3), 241–269 (1998)
23. Sterling, L., Shapiro, E.: The Art of Prolog, 2nd edn. The MIT Press, Cambridge (1994)