

Logic + control: An example of program construction

Włodzimierz Drabent

*Institute of Computer Science, Polish Academy of Sciences,
IDA, Linköpings universitet, Sweden;
drabent at ipipan dot waw dot pl.*

28 July 2012

Abstract

We present a Prolog program (the SAT solver of Howe and King) as a logic program with added control. The control consists of a selection rule (delays of Prolog) and pruning the search space. We construct the logic program together with proofs of its correctness and completeness, with respect to a formal specification. This is augmented by a proof of termination under any selection rule. Correctness and termination are inherited by the Prolog program, the change of selection rule preserves completeness. We prove that completeness is also preserved by one case of pruning; for the other an informal justification is presented.

For proving correctness we use a method, which should be well known but is often neglected. A contribution of this paper is a method for proving completeness. In particular we introduce a notion of semi-completeness, for which a local sufficient condition exists.

We compare the proof methods with declarative diagnosis (algorithmic debugging). We introduce a method of proving that a certain kind of pruning preserves completeness. We argue that the proof methods correspond to natural declarative thinking about programs, and that they can be used, formally or informally, in every-day programming.

KEYWORDS: logic programming, program correctness, program completeness, specification, declarative programming, declarative diagnosis.

1 Introduction

The purpose of this paper is to show to which extent the correctness related issues of a Prolog program can, in practice, be dealt with mathematical precision. We present a construction of a useful Prolog program. We view it as a logic program with added control. We formally prove that the logic program conforms to its specification and partly informally justify that adding control preserves this property. We argue that the employed methods are not difficult and can be used by actual programmers.

Howe and King (2012) presented a SAT solver which is an elegant and concise Prolog program of 22 lines. Formally it is not a (pure) logic program, as it includes `nonvar/1` and the if-then-else of Prolog; it was constructed as an implementation of an algorithm, using logical variables and coroutining. The algorithm is DPLL with watched literals and unit propagation (see (Howe and King 2012) for references). Here we look at the program from a declarative point of view. We show how it can be obtained by adding control to a definite clause logic program.

We first present a simple logic program of five clauses, and then modify it in order

to obtain a logic program on which the intended control can be imposed. The control involves fixing the selection rule (by means of the delay mechanisms of Prolog), and pruning some redundant fragments of the search space. In constructing both the introductory program and the final one, we begin with a specification, describing the relations to be defined by the program. We argue about usefulness of approximate specifications. For both logic programs we present formal proofs of their correctness and completeness. In the second case the proofs are performed together with the construction of the program. We also prove termination under any selection rule. Adding control preserves correctness and termination. Completeness of the final program with control is justified partly informally.

To facilitate the proofs we present the underlying proof methods for correctness and completeness. For proving correctness we use the method of (Clark 1979). For proving completeness we introduce a simplification of the method of (Drabent and Miłkowska 2005). We also introduce a way of proving that a certain kind of pruning SLD-trees preserves completeness. A shorter version of this report is (Drabent 2012).

Preliminaries. In this paper we consider definite clause programs (i.e. logic programs without negation). We use the standard notation and definitions, see e.g. (Apt 1997). In our main examples we assume a Herbrand universe like in Prolog, based on an alphabet of infinitely many function symbols of each arity ≥ 0 . However the theoretical considerations of Sect. 3 are valid for arbitrary nonempty Herbrand universe. By $ground(P)$ we mean the set of ground instances of a program P (under a given Herbrand universe).

We use the Prolog notation for lists. Names of variables begin with an upper-case letter. By a list we mean a term of the form $[t_1, \dots, t_n]$ (so terms like $[a, a|X]$, or $[a, a|a]$ are not considered lists). As we deal with clauses as data, and clauses of programs, the latter will be called *rules* to avoid confusion. Given a predicate symbol p , by an *atom for p* we mean an atom whose predicate symbol is p , and by a *rule for p* – a rule whose head is an atom for p . By a *procedure p* we mean all the rules for p in the program under consideration.

Organization of the paper. The next section presents a simple and inefficient SAT solver. Section 3 is the theoretical part of this paper. It formalizes the notion of a specification, presents the method for proving program correctness, and introduces the methods for completeness. As an example, correctness and completeness of the simple SAT solver are proved. Then related work is discussed, in particular a comparison with declarative diagnosis methods is made. In Section 4 the final logic program is constructed in hand with its correctness and completeness proof. Section 5 considers adding control to the program. Section 6 contains conclusions. The Appendix presents the proofs omitted in Section 3, and a stronger variant of the completeness proving method.

2 Propositional satisfiability – first logic program

Representation of propositional formulae. We first describe the form of data used by the programs discussed in this paper, namely the encoding of propositional formulae in CNF as terms proposed by (Howe and King 2012).

Propositional variables are represented as logical variables; truth values – as con-

stants **true**, **false**. A literal of a clause is represented as a pair of a truth value and a variable; a positive literal, say x , as **true-X** and a negative one, say $\neg x$, as **false-X**. A clause is represented as a list of (representations of) literals, and a conjunction of clauses as a list of their representations. For instance a formula $(x \vee \neg y \vee z) \wedge (\neg x \vee v)$ is represented as $[[\mathbf{true-X}, \mathbf{false-Y}, \mathbf{true-Z}], [\mathbf{false-X}, \mathbf{true-V}]]$. An assignment of truth values to variables can be represented as a substitution. Thus a clause (represented by) f is true under an assignment (represented by) θ iff the list $f\theta$ has an element of the form $t-t$, i.e. **false-false** or **true-true**. A formula in CNF is satisfiable iff its representation has an instance whose each element (is a list which) contains a $t-t$. We will often say “formula f ” for a formula in CNF represented as a term f , similarly for clauses etc.

The program. Now we construct a simple logic program P_1 checking satisfiability of CNF formulae represented as above. We begin with describing the relations (unary relations, i.e. sets) to be defined by the program. Let

$$\begin{aligned} L_1 \text{ be the set of ground terms of the form } [t_1, \dots, t_n|s] \text{ (} n > 0 \text{), where} \\ t_i = t-t \text{ for some } i \in \{1, \dots, n\}, \text{ and some term } t, \quad (1) \\ L_2 \text{ be the set of lists whose all elements are from } L_1. \end{aligned}$$

A clause f is true under an assignment θ iff the list $f\theta$ is in L_1 . A formula in CNF is satisfiable iff it has an instance in L_2 .

Alternatively, we can use a subset $L_1^0 \subseteq L_1$:

$$L_1^0 = \left\{ [t_1-u_1, \dots, t_n-u_n] \mid \begin{array}{l} n > 0, t_1, \dots, t_n, u_1, \dots, u_n \text{ are ground,} \\ t_i = u_i \text{ for some } i \in \{1, \dots, n\} \end{array} \right\}_1$$

and the set L_2^0 of lists whose each element is from L_1^0 . Moreover, any set L_2' such that $L_2^0 \subseteq L_2' \subseteq L_2$ will do²: A formula in CNF is satisfiable iff it has an instance in L_2' (as any its instance from L_2 is also in $L_2' \subseteq L_2$).

We chose L_1, L_2 , as the corresponding program is simpler (and also more efficient). However L_1^0, L_2^0 will be employed in Sect. 4. Predicate *sat_cnf* of the program defines L_2 , it refers to a predicate *sat_cl*, defining L_1 . The program is constructed in a rather obvious way.³

$$\mathit{sat_cnf}([]). \quad (2)$$

$$\mathit{sat_cnf}([Clause|Clauses]) \leftarrow \mathit{sat_cl}(Clause), \mathit{sat_cnf}(Clauses). \quad (3)$$

$$\mathit{sat_cl}([Pol-Var|Pairs]) \leftarrow Pol = Var. \quad (4)$$

$$\mathit{sat_cl}([H|Pairs]) \leftarrow \mathit{sat_cl}(Pairs). \quad (5)$$

$$=(X, X). \quad (6)$$

In the next section we prove that the program defines the intended sets. In Sect. 4 we transform the program into a more sophisticated logic program, for which one can apply the intended control modifications, which result in an efficient Prolog program.

¹ It may be additionally required that $t_u, u_j \in \{\mathbf{true}, \mathbf{false}\}$ for $j = 1, \dots, n$. We do not impose this restriction.

² A wider class of possible sets is shown in (Drabent 2012). (Note that the set L_2 of that paper is a superset of L_2 defined here.)

³ In the rule (4) we followed the style of (Howe and King 2012), the reader may instead prefer a unary rule $\mathit{sat_cl}([Pol-Pol|Pairs])$.

3 Correctness and completeness

In this section we show how to prove that a program indeed defines the required relations. Basically we follow the approach of (Drabent and Milkowska 2005). We present a special case of the correctness criterion used there, and we extend and simplify the method of proving completeness. Then we discuss a certain way of pruning SLD-trees, and introduce a method of proving that the pruning preserves completeness. In this section we allow an arbitrary alphabet of function symbols, requiring only that it contains at least one constant (so the Herbrand universe is nonempty).

Specifications. We provided a specification for the program P_1 by giving a set for each predicate; the predicate should define this set. In a general case, for an n -argument predicate p the specification gives an n -argument relation, to be defined by p . Let us call a ground atom $p(t_1, \dots, t_n)$ **specified** if the tuple (t_1, \dots, t_n) is in the relation corresponding to p . The set S of specified atoms can be seen as a Herbrand interpretation; it is a convenient way to represent the specification. From now on we assume that a (formal) **specification** is a Herbrand interpretation; given a specification S , each $A \in S$ is called a specified atom.

So in our case, the specified atoms are those of the form

$$\begin{aligned} & \text{sat_cnf}(t), \quad \text{where } t \in L_2, \\ & \text{sat_cl}(s), \quad s \in L_1, \\ & x = x, \quad x \text{ is an arbitrary ground term.} \end{aligned} \tag{7}$$

This set of specified atoms will be denoted S_1 .

Correctness and completeness. In imperative 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 relation defined by the program is a subset of the specified one, and completeness means inclusion in the opposite direction. In terms of specified atoms and the least Herbrand model M_P of a program P we have: P is correct w.r.t. S iff $M_P \subseteq S$; it is complete w.r.t. S iff $M_P \supseteq S$ (where S is a specification represented as a set of ground atoms).

It is useful to relate correctness and completeness with answers of programs.⁴

Proposition 1

Let P be a program, Q a query, and S a specification.

If P is correct w.r.t. S and $Q\theta$ is an answer for P then $S \models Q\theta$.

If P is complete w.r.t. S and $S \models Q\sigma$, for a ground $Q\sigma$,⁵ then $Q\sigma$ is an answer for P , and is an instance of some computed answer for P and Q .

⁴ 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 (Apt 1997) 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 $Q\sigma$). Thus, by soundness and completeness of SLD-resolution, $Q\theta$ is an answer for P iff $P \models Q\theta$.

⁵ Note that for any ground query $Q\sigma$ we have $S \models Q\sigma$ iff all the atoms of $Q\sigma$ are in S .

Proof

In the first case, we have $M_P \subseteq S$ and $M_P \models Q\theta$. Hence $S \models Q\theta$. In the second case, $S \subseteq M_P$ and $S \models Q\sigma$, hence $M_P \models Q\sigma$ and thus $P \models Q\sigma$, by Th. 4.30 of (Apt 1997).⁶ By completeness of SLD-resolution, $Q\sigma$ is an instance for some computed answer for P and Q . \square

Approximate specifications. Notice that if a program P is both correct and complete w.r.t. S then $M_P = S$ and the specification describes exactly the relations defined by P . Often it is difficult (and not necessary) to specify the relations exactly. A standard example is the usual definition of *append*, see (Drabent and Miłkowska 2005) for a discussion. In such cases a natural solution is to specify M_P approximately, by giving separate specifications S_{compl}, S_{corr} for completeness and correctness, requiring that $S_{compl} \subseteq M_P \subseteq S_{corr}$. The specifications describe, respectively, which atoms have to be computed, and which are allowed to be computed. We illustrate this approach in Sect. 4, and point out its importance for declarative diagnosis in Sect. 3.4.

3.1 Correctness

To prove correctness we use the following property (Clark 1979); see (Drabent and Miłkowska 2005) for further examples, explanations, references and discussion.

Theorem 2 (Correctness)

A sufficient condition for a program P to be correct w.r.t. specification S is
for each ground instance $H \leftarrow B_1, \dots, B_n$ of a rule of the program,
if $B_1, \dots, B_n \in S$ then $H \in S$.

Note that a compact representation of the sufficient condition is $S \models P$.

Proof

The sufficient condition means that S is a Herbrand model of P . Thus $M_P \subseteq S$, as M_P is the least model of P . \square

Applying Th. 2, it is easy to show that P_1 is correct w.r.t. S_1 . For instance consider rule (5), and its arbitrary ground instance $sat_cl([u|s]) \leftarrow sat_cl(s)$. If $sat_cl(s) \in S_1$ then $s \in L_1$, hence $[u|s] \in L_1$ and $sat_cl([u|s]) \in S_1$. We leave the rest of the proof to the reader.

3.2 Completeness

We begin with introducing a few auxiliary notions. Let us say that a program P is *complete* for an atomic query A if, for any specified ground instance $A\theta$ of A , $A\theta$ is in M_P . Generally, P is **complete for a query** $Q = A_1, \dots, A_n$ w.r.t. a specification S when $S \models Q\theta$ implies that $Q\theta$ is an answer for P , for any ground instance $Q\theta$ of Q (equivalently, $A_1\theta, \dots, A_n\theta \in S$ implies $A_1\theta, \dots, A_n\theta \in M_P$). Informally, complete for Q means that all the answers for Q required by the specification are computed.

⁶ Groundness of $Q\sigma$ is used here, and is necessary for the proposition to hold. As a counterexample, consider a finite alphabet of function symbols, say $\{a, b\}$, and take $P = S$, say $\{p(a), p(b)\}$. Then query $p(X)$ is true in the interpretation S , but it is not an answer for program P (as it is not a logical consequence $P; S \models p(X)$ but $P \not\models p(X)$). However P is complete w.r.t. S .

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 $A \in S$.

We also say that a program P is **semi-complete** w.r.t. S if P is complete for any query Q for which there exists a finite SLD-tree. Note that 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. We also have:

Proposition 3

Let a program P be semi-complete w.r.t. S . The program is complete w.r.t. S if

1. for each ground atomic query $A \in S$ there exists a finite SLD-tree, or
2. the program is recurrent or acceptable (Apt 1997, Chapter 6).

Proof

For a program P semi-complete w.r.t. S , condition 1 implies that P is complete w.r.t. S for each query $A \in S$; hence $S \subseteq M_P$. Condition 2 implies condition 1. \square

A ground atom H is called **covered** (Shapiro 1983) by a program P w.r.t. a specification S if H is the head of a ground instance $H \leftarrow B_1, \dots, B_n$ of a rule of the program, such that all the atoms B_1, \dots, B_n are in S . For instance, given a specification $S = \{p(s^i(0)) \mid i \geq 0\}$, atom $p(s(0))$ is covered both by a program $\{p(s(X)) \leftarrow p(X)\}$ and by $\{p(X) \leftarrow p(s(X))\}$.

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

Theorem 4 (Completeness)

Let P be a definite clause program, S a specification, and Q a query.

If

- all the atoms from S are covered by P , and
- there exists a finite SLD-tree for Q and P

then P is complete for Q w.r.t. S .

If all the atoms from S are covered by P then P is semi-complete w.r.t. S .

The proof and a stronger sufficient condition for completeness are presented in the Appendix.

Let us apply Th. 4 to our program. First let us show that all the atoms from S_1 are covered by P_1 (and thus P_1 is semi-complete). For instance consider a specified atom $A = sat_cnf(t) \in S_1$. Thus t is a ground list of elements from L_1 . If t is nonempty then $t = [s|t']$, where $s \in L_1$, $t' \in L_2$. Thus a ground instance $A \leftarrow sat_cl(s), sat_cnf(t')$ of a rule of P_1 has all its body atoms specified, so A is covered. If t is empty then A is covered as it is the head of the rule $sat_cnf([])$. The reasoning for the remaining atoms of S_1 is similar, and left to the reader.

So the program is semi-complete w.r.t. S_1 , and it remains to show its termination. An informal justification is that, for a reasonable initial query (or for an arbitrary ground initial query), the predicates are invoked with (closed) lists as arguments, and each recursive call employs a shorter list. For a formal proof, we use the standard approach

(Bezem 1993; Apt 1997, Chapter 6.2) and show that the program is recurrent. Let us define a level mapping

$$\begin{aligned} |[h|t]| &= |h| + |t|, \\ |f(t_1, \dots, t_n)| &= 1 \text{ where } n \geq 0 \text{ and } f \text{ is not } [\] , \\ |sat_cnf(t)| &= |sat_cl(t)| = |t|, \\ |t = t'| &= 0, \end{aligned}$$

for any ground terms $h, t, t', t_1, \dots, t_n$, and any function symbol f . Note that $|[t_1, \dots, t_n]| = 1 + \sum_{i=1}^n |t_i|$, and that $|t| > 0$ for any term t . It is easy to show that the program P_1 is recurrent under the level mapping $| \cdot |$, i.e. for each ground instance $H \leftarrow \dots, B, \dots$ of a rule of P_1 , we have $|H| > |B|$. For example, for a ground instance $sat_cnf([t|t']) \leftarrow sat_cl(t), sat_cnf(t')$ of (3) we have $|sat_cnf([t|t'])| = |t| + |t'|$, which is both greater than $sat_cl(t) = |t|$, and than $sat_cnf(t') = |t'|$. (We leave further details to the reader.) By Proposition 3, P_1 is complete w.r.t. S_1 .

In this section we are interested in declarative properties of programs: correctness and completeness. To show completeness of P_1 we proved that it is recurrent. This implies termination for bounded queries, which include ground ones. As a consequence we obtain an important operational property of P_1 : it terminates for the queries for which the program is intended to be used. Consider a query

$$Q = sat_cnf(t) \tag{8}$$

where

$$t \text{ is a list of lists of elements of the form } s-s'. \tag{9}$$

Note that the representations of propositional formulae that we use are of the form (9), and the intended queries to the program are of the form (8). For any ground instance $Q\theta$ of such query $|Q\theta|$ is the same. So Q is bounded. Thus each SLD-tree for P_1 and Q is finite, in other words P_1 terminates for Q under any selection rule.

3.3 Pruning SLD-trees and completeness

Pruning some parts of SLD-trees is often used to improve efficiency of programs. Some kinds of it can be implemented by employing the cut. In our main example we use the if-then-else construct instead, following (Howe and King 2012). 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 show how to formally prove that completeness is preserved under a particular kind of pruning.

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 .

Assume a fixed specification. In the next section we deal with a program P containing two redundant rules, of the form $H \leftarrow B_1, H \leftarrow B_2$. Each of them is sufficient; formally: both $\Pi_1 = P \setminus \{H \leftarrow B_1\}$ and $\Pi_2 = P \setminus \{H \leftarrow B_2\}$ are complete. For a selected atom with the same predicate symbol as H , only one of the rules is to be used. As the choice is dynamic, such pruned SLD-tree is neither an SLD-tree for Π_1 , nor Π_2 . So the

pruned tree may be not complete.⁷ We present a sufficient condition for completeness of such pruned trees.

We will consider SLD-derivations in which not only atoms in the queries, but also program rules are chosen by a selection rule. Let us consider logic programs Π_1, \dots, Π_n ($n > 1$). The intention is that each of them is complete w.r.t. a common specification. Typically, most of the rules of the programs are the same, as in the special case outlined above.

A **csSLD-tree** (cs for clause selection) for a query Q and programs Π_1, \dots, Π_n is constructed as an SLD-tree, but for each node its children are constructed using exactly one program Π_i . (We skip a formal introduction of a notion of cs-selection rule, which selects the program for a node.) Notice that a csSLD-tree for Q and Π_1, \dots, Π_n is a pruned SLD-tree for Q and $\bigcup_i \Pi_i$. An answer of a csSLD-tree is defined in the obvious way.

Proposition 5

Let Π_1, \dots, Π_n be programs, Q a query, and S a specification.

If

for each $i = 1, \dots, n$, all the atoms from S are covered by Π_i , and
a csSLD-tree T for Q is finite

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

For a proof see Appendix A. It immediately follows:

Corollary 6

If each atom from S is covered by each Π_i then whenever $P = \bigcup_i \Pi_i$ terminates for Q under some selection rule R , then each csSLD-tree for Q under R is finite and complete w.r.t. S .

Informally, such csSLD-tree produces all the answers for Q required by S .

3.4 Related work

The correctness proving method of Clark (1979) used here should be well-known, but is often neglected. For instance, an important monograph (Apt 1997) uses a more complicated method, which may be seen as referring to the operational semantics (LD-resolution). Actually, that method proves some property of LD-derivations, from which the declarative property of program correctness follows. See (Drabent and Miłkowska 2005) for comparison and argumentation that the simpler method is sufficient.

Proving completeness has been seldom considered. For instance it is not discussed in (Apt 1997).⁸ Deransart and Małuszyński (1993) present criteria for program completeness, in a sophisticated framework of relating logic programming and attribute

⁷ As an example consider a program P :

$$q(X) \leftarrow p(Y, X). \quad p(Y, 0). \quad p(a, s(X)) \leftarrow p(a, X). \quad p(b, s(X)) \leftarrow p(b, X).$$

where Π_1 , respectively Π_2 are obtained from P by removing one of the last two rules. As a specification for completeness consider $S = \{q(t) \mid t = s^i(0), i \geq 0\}$. Each program P, Π_1, Π_2 is complete, but alternating choice of the last two rules leads to a non complete pruned tree for P .

⁸ Instead, for a program P and an atomic query A , a characterization of the set of computed instances of A is studied, in a special case of the set being finite and the answers ground (Apt 1997, Sect. 8.4). This is based on computing the least Herbrand model (of P or of a certain subset of $ground(P)$).

grammars. Their Th. 6.1 states, roughly speaking, that P is complete w.r.t. S iff there exists a weaker specification S' ($S \subseteq S'$) such that each atom from S' is covered by P w.r.t. S' , and a certain condition holds, which is in a sense similar to P being recurrent (but more general). The method presented here (Th. 4) is a simplification of that from (Drabent and Miłkowska 2005) (an initial version appeared in (Drabent 1999)). In that work the notion of completeness is slightly different, arbitrary interpretation domains are allowed, and a generalization for programs with negation is given. Th. 4 is not a corollary of the results of that work.

Declarative diagnosis. We now discuss the relation between program diagnosis, and proving correctness and completeness of programs. Declarative diagnosis methods (called sometimes declarative debugging) were introduced by Shapiro (1983) (see also (Drabent et al. 1989; Naish 2000) and references therein). They locate in a program the reason for its incorrectness or incompleteness. A diagnosis algorithm begins with a symptom (obtained from testing the program): an answer Q such that $S \not\models Q$, or a query Q for which computation terminates but some answers required by S are not produced. (An alternative notion for incompleteness symptom is an atom $A \in S$ for which the program finitely fails.) The located error turns out to be the program fragment (a rule or a procedure) which violates our sufficient condition for correctness or, respectively, semi-completeness.

More precisely, in declarative diagnosis the reason for incorrectness is an incorrect instance of a program rule. An incorrect rule instance is one which violates the sufficient condition of Th. 2. Obviously, by Th. 2 if the program is incorrect then such rule must exist.⁹

Similarly, as the reason for incompleteness, a diagnosis method finds a not covered specified atom, say $p(\dots) \in S$; in this way procedure p is found to be erroneous. The method is applicable to queries with finite SLD-trees. Existence of a not covered specified atom violates the sufficient condition for completeness of Th. 4. Conversely, if the program is not complete for a query Q with a finite SLD-tree then, by Th. 4, there must exist a not covered specified atom.¹⁰

Another similarity between declarative diagnosis and our proof methods is that the actions performed by a diagnosis algorithm boil down to checking the sufficient conditions for correctness (respectively semi-completeness), but only for some rule instances (some specified atoms) – those involved in producing the symptom.

An attempt to prove a buggy program to be correct (complete) results in violating the corresponding sufficient condition for some rule (specified atom). For instance, in this way the author found an error in a former version of P_1 (there was $[Pairs]$ instead of $Pairs$). Any error located by diagnosis will also be found by a proof attempt;

⁹ However a violation of the sufficient condition does not imply that the program is incorrect. For an example, add all the atoms of the form $sat_cl(t)$ to the specification S_1 , obtaining S'_1 . Program P_1 is still correct w.r.t. S'_1 , but the sufficient condition of Th. 2 does not hold for rule (3). An informal explanation is that S'_1 specifies predicate sat_cl too generally.

Notice that in such case the program is, we may say, potentially incorrect. Replacing some rules by ones satisfying the sufficient condition for correctness (or adding such rules) may result in an incorrect program. For instance, P_1 with an added fact $sat_cl([])$ is incorrect w.r.t. S'_1 .

¹⁰ Again, violating the sufficient condition for completeness does not imply that the program is not complete. (Informally, in such case the specification is not sufficiently general.)

moreover no symptom is needed, and all the errors are found. However the sufficient condition has to be checked for all the rules of the program (for all specified atoms).

A serious difficulty in using declarative diagnosis methods is that they assume an exact specification (a single intended model) of the program. Then answering of some diagnoser queries, like “is *append*([*a*], *b*, [*a|b*]) correct”, may be difficult, as the programmer often does not know some details of the intended model, like those related to applying *append* on non lists. The problem can be overcome by employing approximate specifications (cf. p. 5); using the specification for correctness in incorrectness diagnosis, and that for completeness in diagnosing incompleteness.¹¹

3.5 Discussion

Note that the presented criterion for correctness deals with separate program rules, the criterion for semi-completeness deals with program procedures (to check that an atom $p(\dots)$ is covered one has to consider all the rules for p), and the criteria for completeness involve termination, which may depend on the whole program.

Specifications which are interpretations (as here, in (Apt 1997), and in the declarative diagnosis approaches) have a limitation. They cannot express that e.g. for a given a there exists a b such that $p(a, b)$. A specification has to explicitly state some (one or more) particular b . In our case, we could not specify that it is sufficient for a SAT solver to find *some* variable assignment satisfying f , whenever f is satisfiable. Our specification requires that all such assignments are found. The problem seems to be solved by introducing specifications in a form of logical theories (where axioms like $\exists b.p(a, b)$ can be used). This idea is present in (Deransart and Małuszyński 1993; Drabent and Miłkowska 2005).

Correctness and completeness are declarative properties, they are independent from the operational semantics. If dealing with them required reasoning in terms of operational semantics then logic programming would not deserve to be meant a declarative programming paradigm. The sufficient criteria of Th. 2, 4 for correctness and semi-completeness are purely declarative, they treat program rules as logical formulae, and abstract from any operational notions. The picture is somehow tainted by the step from semi-completeness to completeness. In our approach it involves termination, which is clearly an operational property. Deransart and Małuszyński (1993, Th. 6.1) show how to prove completeness declaratively. Their criterion includes a condition similar to those for proving termination, but seemingly more complicated. Here we chose a simpler solution and refer to program termination, which for practical programs has to be established anyway. Note that semi-completeness alone may be a useful property, as it guarantees that whenever the computation terminates, all the required answers have been computed.

We want to stress the simplicity and naturalness of the sufficient conditions for

¹¹ The problem has been pointed out in (Drabent et al. 1989, Sect. 26.8) and discussed in (Naish 2000) (see also references therein). The solution given in the latter paper is more complicated than what we propose here. A specification in (Naish 2000) classifies each ground atom as *correct*, *erroneous* or *inadmissible*. For such specifications, three-valued declarative debugging algorithms are presented. From our point of view, the set of non-erroneous atoms can be understood as a specification for correctness, and the set of correct atoms as a specification for completeness. However introducing debugging algorithms based on a three-valued logic seems to be an unnecessary complication.

correctness and semi-completeness (Th. 2, 4). 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 has to be the head of a clause instance, whose body atoms should be produced by P too. The author believes that this is a way a competent programmer reasons about (the declarative semantics of) a logic program. The next section illustrates practical applicability of the sufficient conditions in programming.

4 Preparing for adding control

To be able to influence the control of program P_1 in the intended way, in this section we construct a more sophisticated logic program P_3 , with a program P_2 as an initial stage. The construction is guided by a formal specification, and done together with a correctness and semi-completeness proof. We only partially discuss the reasons for particular design decisions in constructing P_3 and in adding control, as the algorithmic and efficiency issues are outside of the scope of this work.

As explained in Sect. 2, it is sufficient that sat_cnf defines an arbitrary set L_{sat_cnf} such that $L_2^0 \subseteq L_{sat_cnf} \subseteq L_2$ (similarly for sat_cl , L_1^0 and L_1). So now we do not specify the set exactly. Instead, in constructing P_2 we will use two specifications: for completeness and for correctness, based on L_1^0, L_2^0 and L_1, L_2 , respectively.

The rules for sat_cnf and $=$ from P_1 , i.e. (2), (3), (6), are included in P_2 . We modify the definition of sat_cl , introducing some new predicates. The new predicates and sat_cl would define the same set L_{sat_cl} (or the subset of L_{sat_cl} of lists longer than 1). However they would represent elements of L_{sat_cl} in a different way.

To simplify the presentation, we provide now the specification for the new predicates. Explanations are given later on, while introducing each predicate. In the specification for correctness the new specified atoms are

$$\begin{aligned} sat_cl3(s, v, p), & \quad \text{where } [p-v|s] \in L_1, \\ sat_cl5(v_1, p_1, v_2, p_2, s), & \\ sat_cl5a(v_1, p_1, v_2, p_2, s), & \quad [p_1-v_1, p_2-v_2|s] \in L_1. \end{aligned} \tag{10}$$

So a specification S_2 for correctness is obtained by adding these literals to specification S_1 . The set of atoms of specification S_2^0 for completeness is described by (7) and (10) with (each occurrence of) L_i replaced by L_i^0 ($i = 1, 2$). Note that $S_2^0 \subseteq S_2$.

In what follows, SC1 stands for the sufficient condition from Th. 2 for correctness w.r.t. S_2 , and SC2 – for the sufficient condition from Th. 4 for semi-completeness w.r.t. S_2^0 (i.e. each atom from S_2^0 is covered). While discussing a procedure p , we consider SC2 for atoms of the form $p(\dots)$ from S_2^0 . Let SC stand for SC1 and SC2.

We leave to the reader a simple check of SC2 for sat_cnf (SC1 for sat_cnf and SC for $=$ have been already done).

Program P_1 performs inefficient search by means of backtracking. We are going to improve it by delaying unification of pairs $Pol-Var$ in sat_cl . The idea is to perform such unification if Var is the only unbound variable of the clause.¹² Otherwise, sat_cl is to be delayed until one of the first two variables of the clause becomes bound to **true** or **false**.

¹² The clause which is (represented as) the argument of sat_cl in the rule for sat_cnf .

This idea will be implemented by separating two cases: the clause has one literal, or more. For efficiency reasons we want to distinguish these two cases by means of indexing the main symbol of the first argument. So the argument should be the tail of the list. (The main symbol is $[]$ for a one element list, and $[\]$ for longer lists.) We redefine sat_cl , introducing an auxiliary predicate sat_cl3 . It defines the same set as sat_cl , but a clause $[Pol-Var|Pairs]$ is represented as three arguments $Pairs, Var, Pol$ of sat_cl3 . A new procedure for sat_cl is obvious:

$$sat_cl([Pol-Var|Pairs]) \leftarrow sat_cl3(Pairs, Var, Pol). \quad (11)$$

SC are trivially satisfied (we leave the simple details to the reader).

Procedure sat_cl3 has to cover each atom $A = sat_cl3(s, v, p) \in S_2^0$, i.e. each A such that $[p-v|s] = [t_1-u_1, \dots, t_n-u_n]$ and $t_i = u_i$ for some i . Assume first $s = []$. Then $p = v$; this suggests a rule

$$sat_cl3([], Var, Pol) \leftarrow Var = Pol. \quad (12)$$

Its ground instance $sat_cl3([], p, p) \leftarrow p = p$ covers A w.r.t. S_2^0 . Conversely, each instance of (12) with the body atom in S_2 is of this form, its head is in S_2 , hence SC1 holds.

When the first argument of sat_cl3 is not $[]$, then we want to delay $sat_cl3(Pairs, Var, Pol)$ until Var or the first variable of $Pairs$ is bound. In order to do this in, say, Sicstus, we need to make the two variables to be separate arguments of a predicate. So we introduce a five-argument predicate sat_cl5 , which is going to be delayed. It defines the set of the lists from L_{sat_cl} of length greater than 1; however a list $[Pol1-Var1, Pol2-Var2|Pairs]$ is represented as the five arguments $Var1, Pol1, Var2, Pol2, Pairs$ of sat_cl5 . The intention is to delay selecting sat_cl5 until its first or third argument is bound (is not a variable). So the following rule completes the definition of sat_cl3 .

$$sat_cl3([Pol2-Var2|Pairs], Var1, Pol1) \leftarrow sat_cl5(Var1, Pol1, Var2, Pol2, Pairs). \quad (13)$$

To check SC, let $S = S_2, L = L_1$ or $S = S_2^0, L = L_1^0$. Then for each ground instance of (13), the body is in S iff the head is in S (as $sat_cl5(v_1, p_1, v_2, p_2, s) \in S$ iff $[p_1-v_1, p_2-v_2|s] \in L$ iff $sat_cl3([p_2-v_2|s], v_1, p_1) \in S$). Hence SC1 holds for (13), and each $sat_cl3([p_2-v_2|s], v_1, p_1) \in S_2^0$ is covered by (13). Thus each $sat_cl3(s', v_1, p_1) \in S_2^0$ is covered by (12) or (13).¹³

In evaluating sat_cl5 , we want to treat the bound variable (the first or the third argument) in a special way. So we make it the first argument of a new predicate sat_cl5a , with the same declarative semantics as sat_cl5 .

$$sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs). \quad (14)$$

$$sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \quad (15)$$

¹³ Notice that some atoms of the form $sat_cl(s), sat_cl3(s, v, p)$ from $S_2 \setminus S_2^0$ are not covered (e.g. when $s = [a, true-true]$); this is the reason why the program is not complete w.r.t. S_2 (and sat_cl in P_2 defines a proper subset of L_1).

SC are trivially satisfied. Moreover, SC2 is satisfied by each of the two rules alone. So each of them is sufficient to define *sat_cl5*. (Formally, the program without (14) or without (15) remains semi-complete.) The control will choose the one that results in invoking *sat_cl5a* with its first argument bound.

To build a procedure *sat_cl5a* we have to provide rules which cover each atom $A = \text{sat_cl5a}(v_1, p_1, v_2, p_2, s) \in S_2^0$. Note that $A \in S_2^0$ iff $[p_1-v_1, p_2-v_2|s] \in L_1^0$ iff $p_1 = v_1$ or $[p_2-v_2|s] \in L_1^0$ iff $p_1 = v_1$ or $\text{sat_cl3}(s, v_2, p_2) \in S_2^0$. So two rules follow

$$\text{sat_cl5a}(\text{Var1}, \text{Pol1}, \text{Var2}, \text{Pol2}, \text{Pairs}) \leftarrow \text{Var1} = \text{Pol1}. \quad (16)$$

$$\text{sat_cl5a}(\text{Var1}, \text{Pol1}, \text{Var2}, \text{Pol2}, \text{Pairs}) \leftarrow \text{sat_cl3}(\text{Pairs}, \text{Var2}, \text{Pol2}). \quad (17)$$

The first one covers A when $p_1 = v_1$, the second when $[p_2-v_2|s] \in L_1^0$. Thus SC2 holds for each atom $\text{sat_cl5a}(\dots) \in S_2^0$. To check SC1, consider a ground instance of (16), with the body atom in S_2 . So it is of the form

$$\text{sat_cl5a}(p, p, v_2, p_2, s) \leftarrow p = p.$$

As the term $[p-p, p_2-v_2|s]$ is in L_1 , the head of the rule is in S_2 . Take a ground instance

$$\text{sat_cl5a}(v_1, p_1, v_2, p_2, s) \leftarrow \text{sat_cl3}(s, v_2, p_2).$$

of (17), with the body atom in S_2 . Then $[p_2-v_2|s] \in L_1$, hence $[p_1-v_1, p_2-v_2|s] \in L_1$, and thus $\text{sat_cl5a}(v_1, p_1, v_2, p_2, s) \in S_2$.

From a declarative point of view, our program is ready. The logic program P_2 consists of rules (2), (3), (6), and (11)–(17). It is correct w.r.t. S_2 and semi-complete w.r.t. S_2^0 .

Avoiding floundering. When selecting *sat_cl5* is delayed as described above, program P_2 may flounder; a nonempty query with no selected atom may appear in a computation.¹⁴ Floundering is a kind of pruning SLD-trees, and may cause incompleteness. To avoid it, we add a top level predicate *sat*. It defines the relation (a Cartesian product) in which the first argument is as defined by *sat_cnf*, and the second argument is a list of truth values (i.e. of **true** or **false**).

$$\text{sat}(\text{Clauses}, \text{Vars}) \leftarrow \text{sat_cnf}(\text{Clauses}), \text{tflist}(\text{Vars}). \quad (18)$$

(Predicate *tflist* will define the set of truth value lists.) The intended initial queries are of the form

$$\text{sat}(f, l), \text{ where } f \text{ is a (representation of a) propositional formula,} \quad (19) \\ l \text{ is the list of variables in } f.$$

Such query succeeds iff the formula f is satisfiable. Floundering is avoided, as *tflist* will eventually bind all the variables of f . More precisely, consider a node Q in an arbitrary SLD-tree for a $\text{sat}(f, l)$ of (19). We have three cases. (i) Q is the root, or its child. (ii) Q contains an atom derived from *tflist*(l). Otherwise, (iii) the variables of l (and thus those of f) are bound; hence all the atoms of Q are ground (as no rule of P_2 introduces a new variable). So no such Q consists solely of non-ground *sat_cl5*

¹⁴ For instance, a query $\text{sat_cnf}([[\text{true}-X, \text{false}-Y]])$ would lead to a query consisting of a single atom $\text{sat_cl5}(X, \text{true}, Y, \text{false}, [])$, which is never selected. On the other hand, a query $\text{sat_cnf}([[\text{true}-X, \text{false}-Y], [\text{false}-X]])$ would lead to selecting $\text{sat_cl3}([], X, \text{false})$, binding X to *false*, and then $\text{sat_cl5}(\text{false}, \text{true}, Y, \text{false}, [])$ is not delayed.

atoms.¹⁵ Moreover, the program has now an additional functionality, as in an answer $sat(f, l)\theta$ the list $l\theta$ represents a variable assignment satisfying f (i -th element of $l\theta$ is the value of the i -th variable of l).

We use auxiliary predicates to define the set of truth values, and of the lists of truth values. The extended formal specification S_3 for correctness consists of atoms

$$\begin{aligned} sat(t, u), & \quad \text{where } t \in L_2, \\ tflist(u), & \quad u \text{ is a list whose elements are true or false.} \\ tf(\mathbf{true}), & \\ tf(\mathbf{false}), & \end{aligned} \quad (20)$$

and those of S_2 (i.e. the atoms of (7), (10)). The extended specification S_3^0 for completeness consists of S_2^0 and of the atoms described by modified (20) where L_2 is replaced by L_2^0 . The three new predicates are defined in a rather obvious way, following (Howe and King 2012):

$$tflist([]). \quad (21)$$

$$tflist([Var|Vars]) \leftarrow tflist(Vars), tf(Var). \quad (22)$$

$$tf(true). \quad (23)$$

$$tf(false). \quad (24)$$

We leave for the reader checking of SC (which is trivial for sat and tf , and rather simple for $tflist$). This completes our construction. The logic program P_3 consists of rules (2), (3), (6), (11) – (18), and (21) – (24). We will also refer to the program with one of the rules for sat_cl5 removed. Let us denote $P_{31} = P_3 \setminus \{(14)\}$ and $P_{32} = P_3 \setminus \{(15)\}$. Each of the three programs is correct w.r.t. S_3 and semi-complete w.r.t. S_3^0 .

Termination of P_3 . To establish completeness of P_3 we show that it is recurrent. Consider a level mapping

$$\begin{aligned} |sat(t, u)| &= \max(3|t|, listsize(u)) + 2, & |sat_cl5(u_1, u_2, u_3, u_4, t)| &= 3|t| + 3, \\ |sat_cnf(t)| &= 3|t| + 1, & |sat_cl5a(u_1, u_2, u_3, u_4, t)| &= 3|t| + 2, \\ |sat_cl(t)| &= 3|t| + 1, & |tflist(u)| &= listsize(u), \\ |sat_cl3(t, u_1, u_2)| &= 3|t| + 1, & |t = u| = |tf(t)| &= 0, \end{aligned}$$

where t, u, u_1, u_2, u_3, u_4 are arbitrary ground terms, $|t|$ is as in Sect. 3.2, and $listsize$ is defined by $listsize([h|t]) = listsize(t) + 1$ and $listsize(f(t_1, \dots, t_n)) = 0$ for any f which is not $[|]$. For example consider (11). For any its ground instance $sat_cl([p-v|t]) \leftarrow sat_cl3(t, v, p)$, the level mapping of the head is $3|t| + 4$, while that of the body atom is $3|t| + 1$. We leave to the reader further details of the proof that P_3 is recurrent.

The program is semi-complete and recurrent, hence it is complete (w.r.t. S_3^0).

As an additional corollary we obtain termination of P_3 under any selection rule for the intended initial queries. Consider a query $Q = sat(t, t')$, where t is a list of lists of elements of the form $s-s'$, and t' is a list. Each intended query to the program is of this form. Q is bounded (for each its ground instance $Q\theta$, $|Q\theta|$ is the same). As P_3 is recurrent, each SLD-tree for P_3 and Q is finite.

¹⁵ Alternatively, non-floundering of this program can be shown automatically, by means of program analysis (King 2012; Genaim and King 2008)

Completeness and pruning. We intend to prune the SLD-trees by using only one of the rules (14), (15) whenever an atom $sat_cl5(\dots)$ is selected. The approach of Sect. 3.3 makes it possible to show that the resulted pruned SLD-trees remain complete w.r.t. S_3^0 . The trees are csSLD-trees for P_{31}, P_{32} . If the root of such tree is Q , as in the previous paragraph, then by Corollary 6 the tree is complete w.r.t. S_3^0 . Informally, all the answers for Q required by the specification are produced by the tree; this is independent from the selection rule.

5 The program with control

In this section we add control to program P_3 . As the result we obtain the Prolog program of Howe and King (2012). (The predicate names differ, those in the original program are related to its operational semantics.) The idea is that P_3 with this control implements the DPLL algorithm with watched literals and unit propagation.¹⁶

The control added to P_3 modifies the default Prolog selection rule (by a `block` declaration), and prunes some redundant parts of the search space (by the if-then-else construct). So correctness and termination of P_3 are preserved (as we proved termination for any selection rule). We introduce two cases of pruning, for the first one we proved that the completeness is preserved. For the second one we justify completeness informally.

The first control feature to impose is delaying sat_cl5 until its first or third argument is not a variable. This can be done by a Sicstus block declaration

$$:- \text{block } sat_cl5(-, ?, -, ?, ?). \quad (25)$$

For the intended initial queries, such delaying does not lead to floundering (as shown in the previous section). So the completeness of the logic program is preserved.

The first case of pruning is using only one of the two rules (14), (15),

$$\begin{aligned} sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) &\leftarrow sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs). \\ sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) &\leftarrow sat_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \end{aligned}$$

the one which invokes sat_cl5a with the first argument bound. We achieve this by employing the `nonvar` built-in and the if-then-else construct of Prolog:

$$\begin{aligned} sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) &\leftarrow \\ &\text{nonvar}(Var1) \rightarrow sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs); \quad (26) \\ &sat_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \end{aligned}$$

Alternatively, the cut could be used, which however seems less elegant. A proof was given in the previous section that this pruning preserves completeness.

An efficiency improvement related to rules (16), (17)

$$\begin{aligned} sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) &\leftarrow Var1 = Pol1. \\ sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) &\leftarrow sat_cl3(Pairs, Var2, Pol2). \end{aligned}$$

is possible. Procedure sat_cl5a is invoked with the first argument $Var1$ bound. If the first argument of the initial query $sat(f, l)$ is a (representation of a) propositional formula then sat_cl5a is called with its second argument $Pol1$ being `true` or `false`. So the unification $Var1 = Pol1$ in (16) works as a test, and the rule binds no variables.¹⁷

¹⁶ However, removing a clause when a literal in it becomes true is implemented only when the literal is watched in the clause.

¹⁷ So `=` may be replaced by the built-in `==`, as in (Howe and King 2012).

Thus after a success of rule (16) there is no point in invoking (17), as the success of (16) produces the most general answer for $sat_cl5a(\dots)$, which subsumes any other answer. Hence the search space can be pruned accordingly. We do this by converting the two rules into

$$\begin{aligned} sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow \\ Var1 = Pol1 \rightarrow true; sat_cl3(Pairs, Var2, Pol2). \end{aligned} \quad (27)$$

This completes our construction.¹⁸ The obtained Prolog program consists of declaration (25), the rules of P_3 except for those for sat_cl5 and sat_cl5a , i.e. (2), (3), (6), (11) – (13), (18), (21) – (24), and Prolog rules (26), (27). It is correct w.r.t. S_3 , and is complete w.r.t. S_3^0 for queries of the form (19).

6 Conclusions

This paper presents proof methods for proving correctness and completeness of definite clause programs, and provides an example of their application: a systematic construction of a Prolog program, the SAT solver of (Howe and King 2012). Starting from a formal specification, a definite clause program, called P_3 , is constructed hand in hand with a proof of its correctness and completeness (Sect. 4). The final Prolog program is obtained from P_3 by adding control (delays and pruning SLD-trees, Sect. 5). Correctness, completeness and termination of a pure logic program can be dealt with formally, and we proved them for P_3 . Adding control preserves correctness and termination (as termination of P_3 is independent from the selection rule). We partly proved, and partly justified informally that completeness is preserved too.

The employed proof methods are of separate interest. The method for correctness (Clark 1979) is simple, should be well-known, but is often neglected. A contribution of this paper is a method for proving completeness (Sect. 3.2), a simplification of that of (Drabent and Miłkowska 2005). It introduces a notion of semi-completeness, for which the corresponding sufficient condition deals with program procedures separately, while for completeness the whole program has to be taken into account. Also a sufficient condition was given that a certain kind of SLD-tree pruning preserves completeness (Sect. 3.3). The methods for proving correctness and semi-completeness are purely declarative, however proving completeness refers to program termination. The reason is that in practice termination has to be concerned anyway, and a pure declarative approach to completeness (Deransart and Małuszyński 1993) seems more complicated (Sect. 3.5).

We point out usefulness of approximate specifications (p. 5). They are crucial for avoiding unnecessary complications in constructing specifications and in correctness and completeness proofs. They are natural: when starting construction of a program,

¹⁸ Employing the cut instead if-then-else, we may obtain the following rules for sat_cl5 , sat_cl5a :

$$\begin{aligned} sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow nonvar(Var1), !, \\ sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs). \\ sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow \\ sat_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \\ sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow Var1 = Pol1, !, \\ sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl3(Pairs, Var2, Pol2). \end{aligned}$$

the relations it should compute are often known only approximately. Also, it is often cumbersome (and unnecessary) to exactly establish the relations computed by a program. As an example and an exercise, the reader may describe the relations defined by the procedures of P_2 .¹⁹

In Sect. 3.4 we compared the proof methods with declarative diagnosis (algorithmic debugging). We showed how approximate specifications lead to avoiding a drawback of declarative diagnosis.

We are interested in declarative programming. Our main example was intended to show how much of the programming task can be done without considering the operational semantics, how “logic” could be separated from “control.” A substantial part of work could be done at the stage of a pure logic program, where correctness, completeness and termination could be dealt with formally. It is important that all the considerations and decisions about the program execution and efficiency (only superficially treated here) are independent from those related to the declarative semantics, to the correctness of the final program, and – to a substantial extent – its completeness.

We argue that the employed proof methods are simple, and correspond to a natural way of declarative thinking about programs (Sect. 3.5). We believe that they can be actually used – maybe at an informal level – in practical programming; this is supported by our main example.

Appendix A

Here we present a proof of Th. 4, a stronger variant (Prop. 8) of this theorem, and a proof of Prop. 5.

Theorem 4 (Completeness)

Let P be a definite clause program, S a specification, and Q a query.

If

- all the atoms from S are covered by P , and
- there exists a finite SLD-tree for Q and P

then P is complete for Q w.r.t. S .

If all the atoms from S are covered by P then P is semi-complete w.r.t. S .

Proof

Assume that all specified atoms (i.e. the atoms in S) are covered, and that a ground query $Q\theta$ consists of specified atoms (i.e. $S \models Q\theta$). For any selection rule R , there exists an SLD-derivation D_R for $Q\theta$ and program $ground(P)$, such that (a) all the queries of D_R consist of specified atoms, and (b) D_R is successful or infinite. By the lifting theorem (Doets 1994, Th. 5.37), it has a lift D'_R , which is an SLD-derivation for Q and P .

So each SLD-tree for Q and P has a branch which is a lift of a derivation of the form D_R above. If the tree is finite then the derivation is finite, hence successful, and all the atoms of $Q\theta$ are in $M_{ground(P)} = M_P$.

This proves the first implication of the theorem, the second one follows immediately.

□

¹⁹ For instance, *sat_cl* defines the set of terms of the form $[t-t]$, or $[t_1-u_1, \dots, t_n-u_n|s]$ ($n > 1$), where $t_i = u_i$ for some i .

We now present an example for which Th. 4 is inapplicable, and introduce a relevant criterion for completeness.

Example 7

Program $P = \{p(s(X)) \leftarrow p(X). p(0). q(X) \leftarrow p(Y).\}$ is complete w.r.t. specification $S = \{p(s^i(0)) \mid i \geq 0\} \cup \{q(0)\}$. It loops for queries $p(X)$ and $q(X)$. Moreover it loops for any instance of $q(X)$. However all the derivations for these queries and program $\text{ground}(P)$ are finite.

Proposition 8 (Completeness)

In Th. 4, condition “there exists a finite SLD-tree for Q and P ” can be replaced by “for each ground instance $Q\theta$ such that $S \models Q\theta$ there exists an SLD-tree for Q and $\text{ground}(P)$ without an infinite branch.”

Proof

The SLD-tree has a branch, which is a derivation D_R as in the proof of Th. 4. The derivation is finite, hence successful, and all the atoms of $Q\theta$ are in $M_{\text{ground}(P)} = M_P$.

□

Proposition 5

Let Π_1, \dots, Π_n be programs, Q a query, and S a specification.

If

- for each $i = 1, \dots, n$, all the atoms from S are covered by Π_i , and
- a csSLD-tree T for Q is finite

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

Proof (outline)

A generalization of the proof of Th. 4. Let Q' be a node of T which has a ground instance $Q'\sigma$ such that $S \models Q'\sigma$. Let the k -th atom A of Q' be selected in Q' . Assume that Π_i is applied to construct the children of Q' . Atom $A\sigma$ is covered by Π_i , let $A\sigma \leftarrow B_1, \dots, B_m$ be a ground instance of a rule from Π_i , with $B_1, \dots, B_m \in S$. Let Q'' be $Q'\sigma$ with $A\sigma$ replaced by B_1, \dots, B_m . Then, by the lifting theorem (Doets 1994, Th. 5.37), Q'' is an instance of a child Q^+ of Q' in T . Obviously, $S \models Q''$.

Let $S \models Q\theta$ for a ground instance $Q\theta$ of Q . By induction, there exists a branch Δ in T such that (1) each its node has a ground instance consisting of atoms from S , (2) the sequence of ground instances is a derivation Γ for $Q\theta$ and $\text{ground}(\bigcup_i \Pi_i)$, and (3) Δ is a lift of Γ . Each nonempty query of Δ has a successor. As T is finite, Δ is a successful derivation (for Q and $\bigcup_i \Pi_i$). By the lifting theorem, $Q\theta$ is an instance of the answer of Δ , which is an answer of T . □

References

- APT, K. R. 1997. *From Logic Programming to Prolog*. International Series in Computer Science. Prentice-Hall.
- BEZEM, M. 1993. Strong termination of logic programs. *J. Log. Program.* 15, 1&2, 79–97.
- CLARK, K. L. 1979. Predicate logic as computational formalism. Tech. Rep. 79/59, Imperial College, London. December.

- DERANSART, P. AND MAŁUSZYŃSKI, J. 1993. *A grammatical view of logic programming*. MIT Press.
- DOETS, K. 1994. *From Logic to Logic Programming*. The MIT Press, Cambridge, MA.
- DRABENT, W. 1999. It is declarative. In *Logic Programming: The 1999 International Conference*. The MIT Press, 607. Poster abstract. A technical report at <http://www.ipipan.waw.pl/~drabent/itsdeclarative3.pdf>.
- DRABENT, W. 2012. Logic + control: An example. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, A. Dovier and V. S. Costa, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 17. 301–311. <http://drops.dagstuhl.de/opus/volltexte/2012/3631>.
- DRABENT, W. AND MIŁKOWSKA, M. 2005. Proving correctness and completeness of normal programs – a declarative approach. *Theory and Practice of Logic Programming* 5, 6, 669–711.
- DRABENT, W., NADJM-TEHRANI, S., AND MAŁUSZYŃSKI, J. 1989. Algorithmic Debugging with Assertions. In *Meta-Programming in Logic Programming*, H. Abramson and M. H. Rogers, Eds. The MIT Press, 501–522.
- GENAIM, S. AND KING, A. 2008. Inferring non-suspension conditions for logic programs with dynamic scheduling. *ACM Trans. Comput. Log.* 9, 3.
- HOWE, J. M. AND KING, A. 2011. A pearl on SAT solving in Prolog (extended abstract). *Logic Programming Newsletter* 24, 1 (March 31). <http://www.cs.nmsu.edu/ALP/2011/03/a-pearl-on-sat-solving-in-prolog-extended-abstract/>.
- HOWE, J. M. AND KING, A. 2012. A pearl on SAT and SMT solving in Prolog. *Theoretical Computer Science*. Special Issue on FLOPS 2010. Available online <http://dx.doi.org/10.1016/j.tcs.2012.02.024>. An earlier version is (Howe and King 2011).
- KING, A. 2012. Private communication.
- NAISH, L. 2000. A three-valued declarative debugging scheme. In *23rd Australasian Computer Science Conference (ACSC 2000)*. IEEE Computer Society, 166–173.
- SHAPIRO, E. 1983. *Algorithmic Program Debugging*. MIT Press.