Logic + control: An example

Włodzimierz Drabent

Institute of Computer Science, Polish Academy of Sciences, Poland, and IDA, Linköping University, Sweden
`drabent@ipipan.waw.pl`

Abstract

We present a Prolog program – the SAT solver of Howe and King – as a (pure) 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. Correctness and termination of the logic program 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. For proving program completeness we employ a new, simpler variant of a method published previously. We point out usefulness of approximate specifications. 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.

1998 ACM Subject Classification D.1.6 Logic Programming, F.3.1 Specifying and Verifying and Reasoning about Programs, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging

Keywords and phrases program correctness, program completeness, specification, declarative programming, declarative diagnosis.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.301

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 [11] presented a SAT solver which is an elegant and concise Prolog program of 22 lines. 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 [11] 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. Both programs terminate 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 [4]; for completeness – a simplification of the method of [8]. We also employ a method of proving that a certain kind of pruning SLD-trees preserves completeness (from [7], an extended version of this paper).

Preliminaries. In this paper we consider definite clause programs (i.e. programs without negation). We use the standard notation and definitions, see e.g. [1]. In our main examples we assume a Herbrand universe $\mathcal{H}$ 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 formalizes the notion of a specification, and presents methods for proving program correctness and completeness. In Section 4 the final logic program is constructed hand in hand with its correctness and completeness proof. Section 5 considers adding control to the program. Section 6 discusses the presented approach and its relation to declarative diagnosis.

2 Propositional satisfiability – first logic program

Representation of propositional formulae. We first present the encoding of propositional formulae in CNF as terms, proposed by [11] and used in this paper.

Propositional variables are represented as logical variables; truth values – as constants `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 `[[true-X,false-Y,true-Z],[false-X,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 checking satisfiability of such formulae. We begin with describing the (unary) relations to be defined by the program. Let

$$L_1^0 = \{\, [t_1\text{-}u_1, \dots, t_n\text{-}u_n] \in \mathcal{H} \mid n > 0,\ t_i = u_i \text{ for some } i \in \{1, \dots, n\} \,\},$$
$$L_2^0 = \{\, [s_1, \dots, s_n] \mid n \geq 0,\ s_1, \dots, s_n \in L_1^0 \,\}.$$

(It may be additionally required that all t_j, u_j are in {`true`, `false`}, we do not impose this restriction). A clause f is true under an assignment θ iff the list $f\theta$ is in L_1^0. A formula in CNF is satisfiable iff it has an instance in L_2^0. However L_2^0 is not a unique set with this property. Moreover, a program defining (exactly) L_2^0 would be unnecessarily complicated and would involve unnecessary computations (like checking if the elements of a list are indeed closed lists of pairs). So we extend L_2^0.

A list of the form $[t_1\text{-}u_1, \ldots, t_n\text{-}u_n]$ $(n \geq 0)$ will be called a *list of pairs*. Let

$$L_1 = \left\{ t \in \mathcal{H} \mid \text{ if } t \text{ is a list of pairs then } t \in L_1^0 \right\},$$
$$L_2 = \left\{ s \in \mathcal{H} \mid \text{ if } s \text{ is a list of lists of pairs then } s \in L_2^0 \right\}.$$

Note that $L_1^0 \subseteq L_1, L_2^0 \subseteq L_2$, and that for any set L_2' such that $L_2^0 \subseteq L_2' \subseteq L_2$ it holds:

$$\text{A formula in CNF is satisfiable iff it has an instance in } L_2'. \tag{1}$$

(Because any its instance from L_2 is also in L_2^0, as the formula is a list of lists of pairs). Thus a program computing any such set L_2' would do.

A program P_1 defining such L_2' is constructed in a rather obvious way. Its main procedure is *sat_cnf*. It employs *sat_cl*, which defines an L_1' such that $L_1^0 \subseteq L_1' \subseteq L_1$. In the next section we prove that the sets defined by the program indeed satisfy these inclusions.

$$\mathit{sat\_cnf}([\,]). \tag{2}$$
$$\mathit{sat\_cnf}([\mathit{Clause}|\mathit{Clauses}]) \leftarrow \mathit{sat\_cl}(\mathit{Clause}),\ \mathit{sat\_cnf}(\mathit{Clauses}). \tag{3}$$
$$\mathit{sat\_cl}([\mathit{Pol}\text{-}\mathit{Var}|\mathit{Pairs}]) \leftarrow \mathit{Pol} = \mathit{Var}. \tag{4}$$
$$\mathit{sat\_cl}([H|\mathit{Pairs}]) \leftarrow \mathit{sat\_cl}(\mathit{Pairs}). \tag{5}$$
$$=(X, X). \tag{6}$$

Let f be (a representation of) a CNF formula. Then $\mathit{sat\_cnf}(f)$ succeeds iff f is satisfiable.

3 Correctness and completeness

Now we show how to prove that a program indeed defines the required relations. Basically we follow the approach of [8]. We present a special case of the correctness criterion used there, and we simplify and extend the method of proving completeness; see [7] for a wider presentation, and for proofs of the theorems.

3.1 Specifications

We provided a specification for the program P_1 by describing a set for each predicate; the predicate should define this set. In a general case, for an n-argument predicate p the specification describes an n-argument relation, to be defined by p. The specification in Sect. 2 is *approximate*: the relations are described not exactly, each one is specified by giving its superset and subset. It is convenient to view an approximate specification as two specifications (in our example the first one specifies L_1, L_2, and the second one L_1^0, L_2^0). The former describes the tuples that are allowed to be computed, the latter those that have to be computed. The former is related to program correctness, the latter to completeness.

In our example it was impossible to provide an exact specification, as it was not known which of the possible relations should be implemented. The usual procedure *append* provides a somehow different example of usefulness of approximate specifications [8]. In that case the relation is known, but it is not necessary (and a bit cumbersome) to specify it exactly.

To make it explicit which relation corresponds to which predicate, specifications will be represented as Herbrand interpretations. A (formal exact) **specification** is a Herbrand interpretation; given a specification S, each $A \in S$ is called a *specified* atom (by S). The fact that $p(t_1, \dots, t_n) \in S$ is understood as that the tuple $(t_1, \dots, t_n)$ is in the relation corresponding to p.

So the approximate specification in the example of Sect. 2 consists of two specifications S_1 and S_1^0 with the specified atoms of the form, respectively:

$$\begin{array}{lllll} S_1\colon & \mathit{sat\_cnf}(t), & \text{where} & t \in L_2, & \\ & \mathit{sat\_cl}(s), & & s \in L_1, & (7)\\ & x = x, & & x \in \mathcal{H} & \end{array} \qquad \begin{array}{lllll} S_1^0\colon & \mathit{sat\_cnf}(t), & \text{where} & t \in L_2^0, & \\ & \mathit{sat\_cl}(s), & & s \in L_1^0, & (8)\\ & x = x, & & x \in \mathcal{H} & \end{array}$$

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. Given a specification S and a program P, with its least Herbrand model M_P, we have: P is **correct** w.r.t. S iff $M_P \subseteq S$; it is **complete** w.r.t. S iff $M_P \supseteq S$.

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. An approximate specification, given by a pair S_{corr}, S_{compl} of Herbrand interpretations, means that for one of them the program has to be correct, for the other – complete. Formally, it is required that $S_{compl} \subseteq M_P \subseteq S_{corr}$.

It is useful to relate correctness and completeness with answers of programs.[1]

▶ 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 $Q\sigma$, then $Q\sigma$ is an answer for P, and is an instance of some computed answer for P and Q.

3.2 Correctness

To prove correctness we use the following property [4]; see [8] for further explanations.

▶ **Theorem 2** (Correctness). *A sufficient condition for a program P to be correct w.r.t. a specification S is $S \models P$.*

Note that $S \models P$ means that for each ground instance $H \leftarrow B_1, \dots, B_n$ of a rule of P, if $B_1, \dots, B_n \in S$ then $H \in S$.

Using 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 $\mathit{sat\_cl}([u|s]) \leftarrow \mathit{sat\_cl}(s)$, such that $\mathit{sat\_cl}(s) \in S_1$. If $[u|s]$ is a list of pairs then s is; thus $s \in L_1^0$, and $[u|s] \in L_1^0$. So $[u|s] \in L_1$, and $\mathit{sat\_cl}([u|s]) \in S_1$. We leave the rest of the proof to the reader.

[1] 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 $Q\sigma$). Thus, by soundness and completeness of SLD-resolution, $Q\theta$ is an answer for P iff $P \models Q\theta$.

3.3 Completeness

We begin with introducing a few auxiliary notions. Let us say that a program P is **complete for a query** $Q = A_1, \ldots, A_n$ w.r.t. S when $A_1\theta, \ldots, A_n\theta \in S$ implies $A_1\theta, \ldots, A_n\theta \in M_P$, for any ground instance $Q\theta$ of Q. Informally, complete for Q means that all the answers for Q required by the specification are computed. 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. Here are conditions under which “semi-complete” implies “complete.”

▶ Proposition 3. Let a program P be semi-complete w.r.t. S. P 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 [1, Chapter 6].

A ground atom H is called **covered** [13] by a program P w.r.t. a specification S if H is the head of a ground instance $H \leftarrow B_1, \ldots, B_n$ of a rule of the program, such that all the atoms $B_1, \ldots, B_n$ 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 program, and S a specification.*
If all the atoms from S are covered by P then P is semi-complete w.r.t. S.

Hence, if such P satisfies one of the conditions from Prop. 3 then it is complete w.r.t. S.

Let us apply Th. 4 to our program. First let us show that all the atoms from S_1^0 are covered by P_1 (and thus P_1 is semi-complete). For instance consider a specified atom $A = sat\_cnf(t)$. Thus $t \in L_2^0$. If t is nonempty then $t = [s|t']$, where $s \in L_1^0$, $t' \in L_2^0$. Hence a ground instance $A \leftarrow sat\_cl(s), sat\_cnf(t')$ of a clause of P_1 has all its body atoms in S_1^0, 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 an intended 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 that the program is recurrent [2],[1, Chapter 6.2], see [7]. Thus by Proposition 3, P_1 is complete w.r.t. S_1.

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. Most of the details are presented. However efficiency issues are outside of the scope of this work.

As explained in Sect. 2, it is sufficient that sat_cnf defines a set L_{sat_cnf} such that $L_2^0 \subseteq L_{sat\_cnf} \subseteq L_2$ (similarly, sat_cl defines L_{sat_cl}, where $L_1^0 \subseteq L_{sat\_cl} \subseteq L_1$). 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 would define the same propositional clauses as sat_cl, but represented 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 (respectively completeness) the new specified atoms are

$$\begin{array}{lll} \mathit{sat\_cl}3(s,v,p), & \text{where} & [p\text{-}v|s] \in L_1 \text{ (respectively} \in L_1^0), \\ \mathit{sat\_cl}5(v_1,p_1,v_2,p_2,s), & & \\ \mathit{sat\_cl}5a(v_1,p_1,v_2,p_2,s), & & [p_1\text{-}v_1,p_2\text{-}v_2|s] \in L_1 \text{ (respectively} \in L_1^0). \end{array} \tag{9}$$

So specification S_2 for correctness is obtained by adding these literals to specification S_1 (cf. (7)), and specification S_2^0 for completeness – by adding to S_1^0 (cf. (8)) the literals of (9) with L_1 replaced by L_1^0. 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 $p(\ldots)$ from S_2^0. Let SC stand for SC1 and SC2. We perform the correctness and completeness proof hand in hand with introducing new rules of P_3. When checking a corresponding SC is not mentioned, it is simple and left to the reader. SC for *sat_cnf* and $=$ have been already shown.

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. Otherwise, *sat_cl* is to be delayed until one of the first two variables of the clause becomes bound to `true` or `false`.

This idea will be implemented by separating two cases: the clause has one literal, or more. 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. We redefine *sat_cl*, introducing an auxiliary predicate *sat_cl*3. It defines the same set as *sat_cl*, but a clause $[Pol\text{-}Var|Pairs]$ is represented as three arguments $Pairs, Var, Pol$ of *sat_cl*3.

$$\mathit{sat\_cl}([Pol\text{-}Var|Pairs]) \leftarrow \mathit{sat\_cl}3(Pairs, Var, Pol). \tag{10}$$

Procedure *sat_cl*3 has to cover each atom $A = \mathit{sat\_cl}3(s,v,p) \in S_2^0$, i.e. each A such that $[p\text{-}v|s] \in L_1^0$. Assume first $s = [\,]$. Then $p = v$; this suggests a rule

$$\mathit{sat\_cl}3([\,], Var, Pol) \leftarrow Var = Pol. \tag{11}$$

Its ground instance $\mathit{sat\_cl}3([\,],p,p) \leftarrow p\!=\!p$ covers A w.r.t. S_2^0. Conversely, each instance of (11) 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_cl*3 is not $[\,]$, we want to delay $\mathit{sat\_cl}3(Pairs, Var, Pol)$ 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_cl*5, which is going to be delayed. It defines the set of the lists from $L_{\mathit{sat\_cl}}$ of length greater than 1; however a list $[Pol1\text{-}Var1, Pol2\text{-}Var2\,|\,Pairs]$ is represented as the five arguments $Var1, Pol1, Var2, Pol2, Pairs$ of *sat_cl*5. The intention is to delay selecting *sat_cl*5 until its first or third argument is bound (is not a variable). So the following rule completes the definition of *sat_cl*3.

$$\mathit{sat\_cl}3([Pol2\text{-}Var2|Pairs], Var1, Pol1) \leftarrow \mathit{sat\_cl}5(Var1, Pol1, Var2, Pol2, Pairs). \tag{12}$$

To check SC, let $S = S_2, L = L_1$ or $S = S_2^0, L = L_1^0$. For each ground instance of (12) the body is in S iff the head is in S. Hence SC1 holds for (12), and each $\mathit{sat\_cl}3([p_2\text{-}v_2|s], v_1, p_1) \in S_2^0$ where $s \neq [\,]$ is covered by (12). So SC2 for *sat_cl*3 holds, due to (11) and (12).

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.

$$sat\_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat\_cl5a(Var1, Pol1, Var2, Pol2, Pairs). \quad (13)$$
$$sat\_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat\_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \quad (14)$$

SC are trivially satisfied. Moreover, SC2 is satisfied by each of the two rules alone. 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 $A = sat\_cl5a(v_1, p_1, v_2, p_2, s) \in S_2^0$. Note that $A \in S_2^0$ iff $[p_1\text{-}v_1, p_2\text{-}v_2|s] \in L_1^0$ iff $p_1 = v_1$ or $[p_2\text{-}v_2|s] \in L_1^0$ iff $p_1 = v_1$ or $sat\_cl3(s, v_2, p_2) \in S_2^0$. So two rules follow

$$sat\_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow Var1 = Pol1. \quad (15)$$
$$sat\_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat\_cl3(Pairs, Var2, Pol2). \quad (16)$$

and SC2 holds for sat_cl5a. To check SC1, consider a ground instance of (15), with the body atom in S_2: $sat\_cl5a(p, p, v_2, p_2, s) \leftarrow p = p$. As $[p\text{-}p, p_2\text{-}v_2|s] \in L_1$, the head of the clause is in S_2. Take a ground instance $sat\_cl5a(v_1, p_1, v_2, p_2, s) \leftarrow sat\_cl3(s, v_2, p_2)$. of (16), with the body atom in S_2. Then its head is in S_2, as $[p_2\text{-}v_2|s] \in L_1$ implies $[p_1\text{-}v_1, p_2\text{-}v_2|s] \in L_1$.

From a declarative point of view, our program is ready. The logic program P_2 consists of rules (2), (3), (6), and (10) – (16). 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. 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.

$$sat(Clauses, Vars) \leftarrow sat\_cnf(Clauses), tflist(Vars). \quad (17)$$

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

$$sat(f, l), \text{ where } f \text{ is a (representation of a) propositional formula, } l \text{ is the list of variables in } f. \quad (18)$$

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 $sat(f, l)$ of (18). 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 atoms.

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

$$\begin{array}{lll} sat(t, u), & tflist(u), & \text{where } t \in L_2,\ u \text{ is a list whose elements are } \texttt{true} \text{ or } \texttt{false}, \\ tf(\texttt{true}), & tf(\texttt{false}), & \end{array} \quad (19)$$

and of those of S_2 (i.e. the atoms of (7), (9)). The extended specification S_3^0 for completeness consists of S_2^0 and of the atoms described by a modified (19) where L_2 is replaced by L_2^0. The three new predicates are defined in a rather obvious way, following [11]:

$$tflist([\,]). \qquad (20) \qquad\qquad tf(true). \qquad (22)$$

$$tflist([Var|Vars]) \leftarrow tflist(Vars), tf(Var). \qquad (21) \qquad\qquad tf(false). \qquad (23)$$

This completes our construction. The logic program P_3 consists of rules (2), (3), (6), (10) – (17), and (20) – (23). It is correct w.r.t. S_3 and semi-complete w.r.t. S_3^0. It terminates for the intended queries, under any selection rule, as it is recurrent under a suitable level mapping, see [7]. Thus by Prop. 3, the program is complete w.r.t. S_3^0.

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 [11]. (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.[2]

The control added to P_3 modifies the default Prolog selection rule, 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).

To delay *sat_cl*5 until its first or third argument is not a variable we use a declaration

```
:- block sat_cl5(-, ?, -, ?, ?).                                  (24)
```

of Sicstus. As informally discussed in Sect. 4, for the intended initial queries floundering is avoided; thus the completeness of P_3 is preserved.

The first case of pruning is to use only one of the two rules (13), (14), the one which invokes *sat_cl5a* with the first argument bound. According to [7, Corollary 6], this pruning preserves completeness (see [7] for a proof). The pruning is implemented by employing the *nonvar* built-in and the if-then-else construct of Prolog:

$$\begin{array}{l} sat\_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow \\ \quad nonvar(Var1) \rightarrow sat\_cl5a(Var1, Pol1, Var2, Pol2, Pairs); \\ \qquad\qquad\qquad\quad sat\_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \end{array} \qquad (25)$$

An efficiency improvement related to rules (15), (16) 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 (15) works as a test, and the rule binds no variables.[3] Thus after a success of rule (15) there is no point in invoking (16), as the success of (15) produces the most general answer for $sat\_cl5a(\ldots)$, which subsumes any other answer. Hence the search space can be pruned accordingly. We do this by converting the two rules into

$$\begin{array}{l} sat\_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow \\ \quad Var1 = Pol1 \ \rightarrow true; \quad sat\_cl3(Pairs, Var2, Pol2). \end{array} \qquad (26)$$

This completes our construction. The obtained Prolog program consists of declaration (24), the rules of P_3 except for those for *sat_cl*5 and *sat_cl5a*, i.e. (2), (3), (6), (10) – (12), (17), (20) – (23), and Prolog rules (25), (26). It is correct w.r.t. S_3, and is complete w.r.t. S_3^0 for queries of the form (18).

[2] However, when a non-watched literal in a clause becomes true, the clause is not immediately removed.
[3] So = may be replaced by the built-in `==`, as in [11].

6 Discussion

Proof methods. The correctness proving method of [4] (further references in [8]) used here should be well-known, but is often neglected. For instance, an important monograph [1] uses a more complicated method (of [3]), which refers to the operational semantics (LD-resolution). See [8] for comparison and argumentation that the simpler method is sufficient.

Proving completeness has been seldom considered, especially within a framework of declarative semantics. For instance it is not discussed in [1]. (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 [1, Sect. 8.4].) Book [5] presents criteria for program completeness, in a sophisticated framework of relating logic programming and attribute grammars. The method presented here (Sect. 3.3, [7]) is a simplification of that from [8] (an initial version appeared in [6]). Our notion of completeness is slightly different, and programs with negation are excluded. We introduce 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.

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. 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 [5, Th. 6.1] seems more complicated [7] (and it includes a condition similar to those for proving termination). 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 correctness and semi-completeness (Th. 2, 4). Informally, the first one says that the rules 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 rule 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.

Specifications. The examples of programs P_1 and P_3 show usefulness of approximate specifications (p. 303). 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, the relations it should compute are often known only approximately. Also, it is often difficult (and unnecessary) to exactly establish the relations computed by a program. As an example, the reader may try to describe the two (distinct) sets defined by the main procedures of P_1 and P_2 (cf. [7], where M_{P_1} is given.)

Specifications which are interpretations (as here and in [1]) have a limitation. They cannot express that e.g. for a given a there exists a b such that $p(a,b)$. 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 specifications S_1^0, S_3^0 require 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 [5, 8].

Relations to declarative diagnosis. Declarative diagnosis methods (called sometimes declarative debugging) [13] (see also [9, 12] and references therein) 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. 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. Roughly speaking, the diagnosis algorithm actually checks the sufficient conditions of Th. 2 (Th. 4), but only for some instances of program rules (for some specified atoms) – those involved in producing the symptom. (See [7] for further discussion.)

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; 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 an exact specification (a single intended model) of the program is needed. Then answering 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 has been pointed out in [9] and discussed in [12] (see also references therein). A solution is to employ approximate specifications; incorrectness diagnosis should use the specification for correctness, and incompleteness diagnosis that for completeness. This seems simpler than introducing new diagnosis algorithms based on three logical values [12].

7 Conclusions

The central part of this paper is an example of a systematic construction of a Prolog program: the SAT solver of [11]. 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 (in this case) termination. We partly proved, and partly justified informally that completeness is preserved too. We point out usefulness of approximate specifications.

The employed proof methods are of separate interest. The method for correctness [4] is simple, should be well-known, but is often neglected. A contribution of this work is a method for proving completeness (Sect. 3.3, [7]), a simplification of that of [8]. Due to lack of space, taking pruning into account in proving completeness [7] is not discussed here.

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. We believe that they can be actually used – maybe at an informal level – in practical programming; this is supported by our main example.

References

1 K. R. Apt. *From Logic Programming to Prolog.* International Series in Computer Science. Prentice-Hall, 1997.

2 M. Bezem. Strong termination of logic programs. *J. Log. Program.*, 15(1&2):79–97, 1993.

3 A. Bossi and N. Cocco. Verifying correctness of logic programs. In J. Díaz and F. Orejas, editors, *TAPSOFT, Vol.2*, volume 352 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 1989.

4 K. L. Clark. Predicate logic as computational formalism. Technical Report 79/59, Imperial College, London, December 1979.

5 P. Deransart and J. Małuszyński. *A grammatical view of logic programming.* MIT Press, 1993.

6 W. Drabent. It is declarative. In *Logic Programming: The 1999 International Conference*, page 607. The MIT Press, 1999. Poster abstract. A technical report at `http://www.ipipan.waw.pl/~drabent/itsdeclarative3.pdf`.

7 W. Drabent. Logic + control: An example of program construction. *CoRR*, arXiv:1110.4978 [cs.LO], 2012. Corrected version. `http://arxiv.org/abs/1110.4978`.

8 W. Drabent and M. Miłkowska. Proving correctness and completeness of normal programs – a declarative approach. *Theory and Practice of Logic Programming*, 5(6):669–711, 2005.

9 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. Algorithmic Debugging with Assertions. In H. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 501–522. The MIT Press, 1989.

10 J. M. Howe and A. King. A pearl on SAT solving in Prolog (extended abstract). *Logic Programming Newsletter*, 24(1), March 31 2011. `http://www.cs.nmsu.edu/ALP/2011/03/a-pearl-on-sat-solving-in-prolog-extended-abstract/`.

11 J. M. Howe and A. King. A pearl on SAT and SMT solving in Prolog. *Theoretical Computer Science*, 2012. Special Issue on FLOPS 2010. Available online `http://dx.doi.org/10.1016/j.tcs.2012.02.024`. An earlier version is [10].

12 L. Naish. A three-valued declarative debugging scheme. In *23rd Australasian Computer Science Conference (ACSC 2000)*, pages 166–173. IEEE Computer Society, 2000.

13 E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.