TLP **18** (1): 1–29, 2018. © Cambridge University Press 2017
doi:10.1017/S1471068417000047 First published online 19 June 2017

Logic + control: On program construction and verification

WŁODZIMIERZ DRABENT
Institute of Computer Science, Polish Academy of Sciences, Warsaw
IDA, Linköping University, Linköping, Sweden
(*e-mails*: drabent@ipipan.waw.pl)

submitted 2 June 2016; revised 18 April 2017; accepted 18 April 2017

Abstract

This paper presents an example of formal reasoning about the semantics of a Prolog program of practical importance (the SAT solver of Howe and King). The program is treated as a definite clause logic program with added control. The logic program is constructed by means of stepwise refinement, hand in hand with its correctness and completeness proofs. The proofs are declarative – they do not refer to any operational semantics. Each step of the logic program construction follows a systematic approach to constructing programs which are provably correct and complete. We also prove that correctness and completeness of the logic program is preserved in the final Prolog program. Additionally, we prove termination, occur-check freedom and non-floundering.

Our example shows how dealing with "logic" and with "control" can be separated. Most of the proofs can be done at the "logic" level, abstracting from any operational semantics.

The example employs approximate specifications; they are crucial in simplifying reasoning about logic programs. It also shows that the paradigm of semantics-preserving program transformations may be not sufficient. We suggest considering transformations which preserve correctness and completeness with respect to an approximate specification.

KEYWORDS: logic programming, declarative programming, program completeness, program correctness, specification, program transformation, floundering, occur-check

1 Introduction

The purpose of this paper is to show that (1) the correctness-related issues of Prolog programs can, in practice, be dealt with mathematical precision and (2) most of the reasoning can be declarative (i.e. not referring to any operational semantics, in other words depending only on the logical reading of programs). We present a construction of a useful Prolog program. We view it as a logic program with added control (Kowalski 1979). The construction of the logic program is guided by (and done together with) a proof that the program conforms to its specification. The Prolog program is obtained from the logic program by adding control. We prove that adding control preserves the conformity with the specification. The proofs follow the approach presented and discussed in Drabent (2016a). We believe that the employed proof methods are not difficult and can be used in actual practical programming.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

The notion of partial correctness of programs (in imperative and functional programming) divides in logic programming into correctness and completeness. Correctness means that all answers of the program are compatible with the specification, completeness – that the program produces all the answers required by the specification. A specification may be *approximate*: for such a specification, some answers are allowed, but not required to be computed. The program construction presented in this paper illustrates the usefulness of approximate specifications.

For proving correctness, we use the method of Clark (1979). The method should be well known, but is often neglected. For proving completeness, a method of Drabent (2016a) is used. It introduces a notion of *semi-completeness*; semi-completeness and termination imply completeness. We also employ an approach from Drabent (2016a) for proving that completeness is preserved under pruning of SLD-trees. We use the sufficient condition for termination of Bezem (1993), and introduce a sufficient condition for non-floundering.

We are interested in treating logic programming as a declarative paradigm, and in reasoning about programs declaratively, i.e. independently from their operational semantics. Correctness, semi-completeness and completeness are declarative properties of programs. To prove them, this paper uses purely declarative methods. The employed sufficient condition for completeness of pruned SLD-trees abstracts, to a substantial extent, from details of the operational semantics.

The program dealt with in this paper is the SAT solver of Howe and King (2012). It is an elegant and concise Prolog program of 22 lines. Formally, it is not a logic program, as it includes `nonvar/1` and the if-then-else construct of Prolog; it was constructed as an implementation of an algorithm, using logical variables and coroutining. The algorithm is DPLL (Davis et al. 1962), with watched literals and unit propagation (see Gomes et al. (2008), Howe and King (2012) and references therein). 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 four clauses, and then modify it (in two steps) in order to obtain a logic program on which the intended control can be imposed. The construction of each program begins with a specification, describing the relations to be defined by the program. The construction is guided by a proof of the program's correctness and semi-completeness, and is performed hand in hand with the proof. We formulate a general approach to such program construction. The control imposed on the last logic program involves modifying the selection rule (by means of delay declarations) and pruning some redundant fragments of the search space. Such control preserves correctness, and we prove that completeness is also preserved (which in general may be violated by pruning, or by floundering).

It is important that logic and control are separated in the construction. Most of the work has been done at the level of logic programs. Their correctness and completeness could be treated formally, independently from the operational semantics. All the considerations related to the operational semantics, program behaviour and efficiency are independent from those related to the declarative semantics, correctness and completeness.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Related work. For an overview of work on proving correctness and completeness of logic programs, see Drabent (2016a), Drabent and Miłkowska (2005). In particular, little work has been done on program completeness. The author is not aware of published correctness (or completeness) proofs that are declarative, and deal with practical logic programs (not smaller than the program dealt with here)[1]. A possible exception is a correctness proof of a toy compiler of seven clauses (Deransart and Małuszyński 1993).

A preliminary version of this article was a conference paper (Drabent 2012); it presented basically the same program construction, however, some proofs presented here were missing. In particular, preserving completeness under pruning was dealt with only informally. The methods for dealing with completeness and correctness are presented and discussed elsewhere (Drabent 2016a); here, we augment that paper by a more substantial example.

Our approach differs from semantics-preserving program transformations (Pettorossi et al. 2010, and the references therein). The latter is focused on formal steps for transforming programs into more efficient ones with the same semantics (of their main predicates). The initial program is treated as a specification. In our example, the role of intermediate programs is to illustrate the informal development process, and support explaining the design decisions. Correctness and completeness are considered for each program separately. An interesting feature of our example is that the consecutive programs are not equivalent – their main predicates define different relations, satisfying however the same approximate specification. We argue (Section 4.5) that the paradigm of semantics-preserving transformations is inapplicable to this case. This suggests that it should be useful to generalize the paradigm to transformations which preserve correctness and completeness w.r.t. an approximate specification.

Preliminaries. This paper considers definite clause logic programs (then implemented as Prolog programs). We use the standard notation and definitions (Apt 1997). 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 a *p-atom* (or *atom for p*) we mean an atom whose predicate symbol is p, and by a *rule for p* – a rule whose head is a p-atom. The set of the rules for p in the program under consideration is called *procedure p*.

We assume a fixed alphabet of function and predicate symbols. The alphabet may contain symbols not occurring in the considered program. The Herbrand universe (the set of ground terms) will be denoted by $\mathscr{HU}$, the Herbrand base (the set of ground atoms) by $\mathscr{HB}$. For an expression (a program) E, by $ground(E)$ we mean the set of ground instances of E (ground instances of the rules of E). $\mathscr{M}_P$ denotes

[1] The proof methods employed by Apt (1997) and by Pedreschi and Ruggieri (1999) are not declarative. In particular, they depend on the order of atoms in clause bodies. They prove certain properties of LD-derivations, from which program correctness, in the sense considered here, follows. (In the case of the latter paper, also program completeness can be concluded.)

Similarly, the approach of Deville (1990) is not declarative. Also, it deals with constructing programs from specifications; correctness and completeness follow from construction. Thus, reasoning about correctness or completeness of arbitrary programs is not dealt with.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

the least Herbrand model of a program P. An expression is *linear* if every variable occurs in it at most once.

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. (So we use "computed/correct answer" instead of "computed/correct instance of a query" of Apt (1997).) 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 instance of Q). Thus, by soundness and completeness of SLD-resolution, $Q\theta$ is an answer for P iff $P \models Q\theta$.

We also deal with a generalization of standard SLD-resolution, allowing the selection rule to be a partial function (a *partial selection rule*). Thus, in some queries, no atom may be selected, such a query is called *floundered*. Obviously, such a generalization preserves correctness, but not completeness of SLD-resolution. By the Prolog selection rule, we mean selecting the first atom in each query. (So most of Prolog implementations also implement other selection rules; this is called delays or coroutining.) LD-resolution means SLD-resolution under the Prolog selection rule.

By "declarative" (property, reasoning, ...), we mean referring only to the logical reading of programs, thus abstracting from any operational semantics. So Q being an answer for P is a declarative property. In particular, properties depending on the order of atoms in rules will not be considered declarative, as the logical reading does not distinguish equivalent formulae, like $\alpha \wedge \beta$ and $\beta \wedge \alpha$.

Names of variables begin with an upper-case letter. We use the list notation of Prolog. So $[t_1, \ldots, t_n]$ $(n \geqslant 0)$ stands for the list of elements $t_1, \ldots, t_n$. Only a term of this form is considered a list. (Thus, terms like $[a, a|X]$ or $[a, a|a]$, where a is a constant distinct from [], are not lists). The set of natural numbers will be denoted by $\mathbb{N}$.

Outline of the paper. Sections 2 and 5 present theoretical results showing how to, respectively, prove correctness and completeness of programs, and prove that completeness is preserved under pruning of SLD-trees. Sections 3 and 4 develop a SAT-solver as a logic program, hand in hand with its correctness and completeness proof. Additionally, the latter section proves non-floundering and occur-check freedom, and discusses semantics-preserving transformations in the context of our example. Section 6 converts the logic program into a Prolog program with the intended control imposed. It also shows that the Prolog program preserves the correctness and completeness of the logic program. The last section contains a discussion.

2 Correctness and completeness of programs

This section introduces the notions of specification, correctness and completeness. Then it presents a way of proving that definite logic programs are correct and complete. The approach is declarative. It does not depend on any operational semantics, and programs are viewed as sets of logic formulae. In particular, the reasoning is independent from the order of atoms in rules. We conclude with

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

presenting a way of constructing programs which are provably correct and semi-complete. The presentation here is rather brief, for a more comprehensive treatment, see Drabent (2016a). This section lacks examples, as the concepts introduced here are employed later on in the paper. In particular, Section 3 provides examples for most of the definitions and results presented here.

2.1 The notions

Specifications. From a declarative point of view, logic programs compute relations. A specification should describe these relations. It is convenient to assume that the relations are over the Herbrand universe. A handy way for describing such relations is a Herbrand interpretation; it describes, as needed, a relation for each predicate symbol of the program.

Definition 1

By a **specification**, we mean a Herbrand interpretation, i.e. a subset of $\mathscr{HB}$.

The relation described by a specification S for a predicate p of arity n is $\{(t_1,\ldots,t_n) \mid p(t_1,\ldots,t_n) \in S\}$. The atoms from a specification S will be called the **specified** atoms (by S).

Obviously, the relations actually defined by a program P are described by its least Herbrand model $\mathscr{M}_P$; the relation for a predicate p is $\{(t_1,\ldots,t_n) \mid p(t_1,\ldots,t_n) \in \mathscr{M}_P\}$.

Correctness and completeness. In imperative and functional programming, (partial) correctness usually means that the program results are as specified (provided the program terminates). In logic programming, due to its non-deterministic nature, we actually have two issues: *correctness* (all the results are compatible with the specification) and *completeness* (all the results required by the specification are produced). In other words, correctness means that the relations defined by the program are subsets of the specified ones, and completeness means inclusion in the opposite direction. Formally:

Definition 2

Let P be a program and $S \subseteq \mathscr{HB}$ a specification. P is **correct** w.r.t. S when $\mathscr{M}_P \subseteq S$; it is **complete** w.r.t. S when $\mathscr{M}_P \supseteq S$.

We will sometimes skip the specification when it is clear from the context. For a program P correct w.r.t. S, if a query Q is an answer of P, then $S \models Q$. (Remember that Q is an answer of P iff $P \models Q$.) When P is complete w.r.t. S and Q is ground, then $S \models Q$ implies that Q is an answer of P. More generally, this implication holds for arbitrary queries such that $\mathscr{M}_P \models Q$ implies $P \models Q$ (for sufficient conditions, see Maher (1988), Apt (1997), Drabent (2016a, 2016b)). In particular, the implication holds when the alphabet of function symbols is infinite.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

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

Definition 3
A **predicate** p in P is **correct** w.r.t. S when each p-atom of $\mathscr{M}_P$ is in S, and **complete** w.r.t. S when each p-atom of S is in $\mathscr{M}_P$.

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

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

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

Approximate specifications. It happens quite often in practice that a programmer does not know exactly the relations defined by a program and, moreover, such knowledge is unnecessary. For example, it is irrelevant whether $append([a], 1, [a|1])$ is an answer of a list appending program or not. In such cases, it is sufficient to specify the program's semantics approximately. More formally, to provide distinct specifications, say S_{compl} and S_{corr}, for completeness and correctness of the considered program P. The intention is that $S_{\text{compl}} \subseteq \mathscr{M}_P \subseteq S_{\text{corr}}$. So the specification for completeness says what the program has to compute, and the specification for correctness – what it may compute; in other words, the program should not produce any answers incorrect w.r.t. the specification for correctness. It is irrelevant whether atoms from $S_{\text{corr}} \setminus S_{\text{compl}}$ are, or are not, answers of the program. For example, the standard APPEND program does not define the list concatenation relation, but its superset (and $append([a], 1, [a|1])$ is an answer of the program, but $append([a], a, [a|1])$ is not). In this case, S_{compl} describes the list concatenation relation; see Drabent (2016a) for S_{corr} and further discussion.

Definition 4
An **approximate specification** is a pair $S_{\text{compl}}, S_{\text{corr}}$ of specifications, where $S_{\text{compl}} \subseteq S_{\text{corr}} \subseteq \mathscr{HB}$.

A program P is **fully correct** w.r.t. $S_{\text{compl}}, S_{\text{corr}}$ when $S_{\text{compl}} \subseteq \mathscr{M}_P \subseteq S_{\text{corr}}$. A predicate p in P is fully correct w.r.t. $S_{\text{compl}}, S_{\text{corr}}$ when each p-atom of S_{compl} is in $\mathscr{M}_P$, and each p-atom of $\mathscr{M}_P$ is in S_{corr}.

Correctness (respectively completeness) w.r.t. $S_{\text{compl}}, S_{\text{corr}}$ will mean correctness w.r.t. S_{corr} (completeness w.r.t. S_{compl}). By abuse of terminology, a single specification may be called approximate when the intention is that the specification is distinct from $\mathscr{M}_P$.

In many cases, the atoms from $S_{\text{corr}} \setminus S_{\text{compl}}$ may be (formally or informally) considered as ill-typed. For instance, in $append([a], 1, [a|1])$ terms 1 and $[a|1]$ are not lists. In the context of typed logic programming (Pfenning 1992), the need for approximate specifications may disappear. For instance, in an appropriate typed logic, the standard APPEND program may define the list concatenation exactly. However in practice, we usually deal with untyped logic programming. In particular, the programming language Prolog is untyped. See Drabent (2016a) for further discussion, including the need for approximate specifications in typed logic programming.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

The main example of this paper employs approximate specifications. In particular, various versions of the constructed program define different relations for the same predicate symbols, but the programs are correct and complete w.r.t. the same approximate specification. (More precisely, their common predicates are fully correct w.r.t. the same approximate specification.) The example shows that employing approximate specifications results in simplifying the construction of specifications and proofs. It is useful (Drabent 2016a) in declarative diagnosis (also known as algorithmic debugging (Shapiro 1983)). For further discussion and examples, see Drabent (2016a), Drabent and Miłkowska (2005).

2.2 Reasoning about correctness

The following sufficient condition for program correctness will be used.

Theorem 5
Let P be a program and S be a specification. If $S \models P$, then P is correct w.r.t. S.

In other words, the sufficient condition for correctness of P is that for each ground instance $H \leftarrow B_1, \ldots, B_n$ of a rule of P, if $B_1, \ldots, B_n \in S$, then $H \in S$.

Deransart (1993) attributes this result to Clark (1979). It should be well known, but is often unacknowledged. Often more complicated correctness proving methods, based on the operational semantics, are proposed, e.g. by Apt (1997). See Drabent and Miłkowska (2005), Drabent (2016a) for further comparison, examples and discussion.

2.3 Reasoning about completeness

Little work has been devoted to reasoning about completeness of programs. See Drabent (2016a) for an overview. We summarize the approach of Drabent (2016a), stemming from that of Drabent and Miłkowska (2005). It is based on an auxiliary notion of semi-completeness.

Definition 6
A program P is **semi-complete** w.r.t. a specification S if P is complete w.r.t. S for any query Q such that there exists a finite SLD-tree for P and Q.

Less formally, the existence of a finite SLD-tree means that P with Q terminates under some selection rule ($\exists$-terminates in the terms of Pedreschi et al. (2002)). For a semi-complete program P, if a computation for a query Q terminates, then all the answers for Q required by the specification have been obtained. So establishing completeness is divided into showing completeness and termination. Obviously, a complete program is semi-complete. We immediately obtain:

Proposition 7 (*completeness*)
Assume that a program P is semi-complete w.r.t. a specification S. Then, P is complete w.r.t. S if

(1) each atom $A \in S$ is a root of a finite SLD-tree for P, or
(2) each atom $A \in S$ is an instance of an atom Q which is a root of a finite SLD-tree for P.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Our sufficient condition for semi-completeness employs the following notion, stemming from Shapiro (1983).

Definition 8
A ground atom H is **covered by a rule** C w.r.t. a specification S if H is the head of a ground instance $H \leftarrow B_1, \ldots, B_n$ ($n \geqslant 0$) of C, such that all the atoms $B_1, \ldots, B_n$ are in S.

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

Informally, H covered by C w.r.t. S means that C can produce H out of the atoms in S. The following sufficient condition provides a method of proving semi-completeness.

Theorem 9 (*semi-completeness* (Drabent 2016a))
If all the atoms from a specification S are covered w.r.t. S by a program P, then P is semi-complete w.r.t. S.

We will say that a procedure p of P *satisfies the sufficient condition for semi-completeness* w.r.t. S if each p-atom of S is covered by P w.r.t. S.

To prove program completeness, the theorem has to be augmented by a way of proving termination. Fortunately, for the purposes of this paper, a simple method of Bezem (1993) is sufficient.

Definition 10
A **level mapping** is a function $|\ |: \mathscr{HB} \to \mathbb{N}$ assigning natural numbers to ground atoms.

A program P is **recurrent** w.r.t. a level mapping $|\ |$ (Bezem 1993; Apt 1997) if, in every ground instance $H \leftarrow B_1, \ldots, B_n \in \mathit{ground}(P)$ of its rule ($n \geqslant 0$), $|H| > |B_i|$ for all $i = 1, \ldots, n$.

A program is recurrent if it is recurrent w.r.t. some level mapping. A rule C is recurrent (w.r.t. $|\ |$) if program $\{C\}$ is recurrent (w.r.t. $|\ |$).

A query Q is **bounded** w.r.t. a level mapping $|\ |$ if, for some $k \in \mathbb{N}$, $|A| < k$ for each ground instance A of an atom of Q.

Theorem 11 (*Termination* (Bezem 1993))
Let P be a program recurrent w.r.t. a level mapping $|\ |$, and Q be a query bounded w.r.t. $|\ |$. Then all SLD-derivations for P and Q are finite.

From the theorem and Proposition 7, we immediately obtain:

Corollary 12 (*completeness*)
If a program P is semi-complete w.r.t. a specification S and recurrent, then P is complete w.r.t. S.

Note that the sufficient conditions for correctness (Theorem 5) and semi-completeness (Theorem 9) are declarative. So is the sufficient condition for completeness of Corollary 12. Thus, the correctness and completeness proofs presented later on in this paper are declarative. In general, the presented approach

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

for proving completeness is not declarative, as termination (finiteness of SLD-trees) depends on the selection rule. For instance, the well-known method of Apt and Pedreschi (1993) of proving termination, depends on the order of atoms in program rules.

We mention another declarative way of showing program completeness (Deransart and Małuszyński 1993) (see also (Drabent 2016a)), applicable also to non-terminating programs. In that approach, a level mapping is employed, however, it may be defined only on atoms from the specification S.

Theorem 13 (*completeness*)
Let P be a program, S a specification and $|\ |: S \to \mathbb{N}$. If each atom $A \in S$ is covered w.r.t. S by some ground instance $A \leftarrow B_1, \ldots, B_n \in \mathit{ground}(P)$ such that $|A| > |B_i|$ for $i = 1, \ldots, n$, then P is complete w.r.t. S.

Note the similarity of this condition to that of Theorem 9 together with Corollary 12. The difference is that here only a fragment of $\mathit{ground}(P)$ is required to be recurrent. The fragment consists of a rule instance covering A for each $A \in S$. Due to this similarity, the amount of work to prove program completeness by Theorem 13 is often similar to that of proving it by showing semi-completeness and termination. So, in practice, the latter is usually preferable, as the termination has to be shown anyway (under the selection rule used by the implementation of the program).

2.4 Program construction

The presented sufficient conditions suggest a systematic (informal) method of constructing programs which are provably correct and semi-complete. A guiding principle is that the program should satisfy the sufficient condition for semi-completeness. The construction results in a program together with proofs of its correctness and semi-completeness.

Assume that specifications S_{compl} and S_{corr} are given for, respectively, completeness and correctness of a program P to be built. For each predicate p occurring in S_{compl}, consider the set $S_p = \{\, p(\bar{t}) \mid p(\bar{t}) \in S_{\text{compl}} \,\}$ of the specified p-atoms from the specification. To construct a procedure p of P, provide rules such that

(1) each atom $A \in S_p$ is covered w.r.t. S_{compl} by some rule, and
(2) each rule satisfies the sufficient condition for correctness w.r.t. S_{corr} of Theorem 5.

(In other words, the first requirement states that the constructed procedure satisfies the sufficient condition for semi-completeness.) The constructed program P is the union of the procedures for all p from S_{compl}. It satisfies the conditions of Theorems 5, 9. Thus, P is correct w.r.t. S_{corr} and semi-complete w.r.t. S_{compl}.

In practice, semi-completeness is not sufficient. The actual task is to obtain a program which is complete; also in most cases, the program should terminate for the intended class of initial queries. So the constructed program rules should additionally satisfy some sufficient condition for completeness (like that of Theorem 13) or for

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

termination (like the program being recurrent). In this way, the method of the previous paragraph may be augmented to ensure not only correctness and semi-completeness, but also completeness.

The approach presented here will be used throughout the program constructions presented in this paper.

3 SAT solver – first logic program

We are ready to begin the main subject of this paper – a construction of a program implementing a SAT solver. The construction is divided in several steps, three definite clause logic programs and a final Prolog program are constructed. An interesting feature is that the construction is not a case of semantics-preserving program transformation. The programs define different relations (for the common predicates); however, the common predicates are correct and complete w.r.t. the same approximate specification.

This section explains the data structures used by the programs, provides a specification, and presents a construction of the first program, hand in hand with a correctness and semi-completeness proof.

Representation of propositional formulae. We first describe the form of data used by our programs, 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 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 a term) s is true under an assignment (represented by) θ iff the list $s\theta$ 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 elements (being lists) contain a t-t each. We will often say "formula u" for a formula in CNF represented as a term u, similarly for clauses, etc.

Specification. Now, let us describe the sets to be defined by the predicates of our first SAT-solving program.

$$\begin{aligned} L_1 &= \{\, [t_1,\ldots,t_i|s] \in \mathscr{HU} \mid i > 0,\ t_i = t\text{-}t \text{ for some } t \in \mathscr{HU} \,\}, \\ L_2 &= \{\, [s_1,\ldots,s_n] \in \mathscr{HU} \mid n \geqslant 0,\ s_1,\ldots,s_n \in L_1 \,\}. \end{aligned} \tag{3.1}$$

Note that $t_1,\ldots,t_{i-1},s$ are arbitrary ground terms; the reason for such generality will be explained further on. The following holds for L_1 and L_2.

A clause s is true under an assignment θ iff the list $s\theta$ is in L_1. Hence,
a CNF formula u is true under θ iff $u\theta$ is in L_2, (3.2)
a CNF formula u is satisfiable iff u has an instance in L_2.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Let us require that predicates *sat_cl*, *sat_cnf* define in our program the sets L_1, L_2. So the specification is

$$S_1 = \{\, sat_cl(s) \mid s \in L_1 \,\} \cup \{\, sat_cnf(u) \mid u \in L_2 \,\}.$$

From (3.2), it follows that each program which is correct and complete w.r.t. S_1 works as a SAT solver: for any CNF formula u and any (substitution representing an) assignment θ,

$$\begin{array}{l} u \text{ is true under } \theta \text{ iff } \theta \text{ is an answer substitution for } sat_cnf(u), \\ u \text{ is satisfiable iff query } sat_cnf(u) \text{ succeeds.} \end{array} \tag{3.3}$$

We are ready with the specification for our first program. However, it is not necessary that a SAT-solving program defines the set L_2. Another choice may be

$$\begin{array}{l} L_1^0 = \left\{\, [t_1\text{-}t_1', \ldots, t_n\text{-}t_n'] \in \mathscr{HU} \;\middle|\; n > 0,\ t_i = t_i' \text{ for some } i \in \{1, \ldots, n\} \,\right\}, \\ L_2^0 = \{\, [s_1, \ldots, s_n] \in \mathscr{HU} \mid n \geqslant 0,\ s_1, \ldots, s_n \in L_1^0 \,\}, \end{array} \tag{3.4}$$

as (3.2) holds for L_1^0, L_2^0. (Note that $L_1^0 \subseteq L_1$ and $L_2^0 \subseteq L_2$; for simplicity, we allow arbitrary terms t_i, t_i', not only `false`, `true`.) This leads to a specification

$$S_1^0 = \{\, sat_cl(s) \mid s \in L_1^0 \,\} \cup \{\, sat_cnf(u) \mid u \in L_2^0 \,\}.$$

Moreover, any sets L_1', L_2' such that $L_1^0 \subseteq L_1' \subseteq L_1$ and $L_2^0 \subseteq L_2' \subseteq L_2$ will do, as such L_1', L_2' satisfy (3.2)[2]. Hence, any program complete w.r.t. S_1^0 and correct w.r.t. S_1 has property (3.3), and thus can be used as a SAT solver.

We have chosen L_2 as the set defined by our first program (so S_1 is its specification both for correctness and for completeness). This leads to a simpler program – a check is avoided that certain terms are lists, and that their elements are of the form $t\text{-}t'$. However, our further programs will define a set L_2' as above, employing S_1^0, S_1 as an approximate specification (for predicates *sat_cl*, *sat_cnf*)[3].

The first program. Now we construct a program P_1, hand in hand with its correctness and semi-completeness proofs. We follow the program construction approach described in Section 2.4. So we construct program rules such that each atom from S_1 is covered w.r.t. S_1 by some of the rules. Additionally, care is taken that the rules are recurrent. (This is not made explicit in our text.) For each rule, the sufficient condition for correctness of Theorem 5 will be checked. At the end, we prove that the constructed program is recurrent, hence complete and terminating.

[2] Because if a clause s has an instance $s\theta \in L_1$ then $s\theta \in L_1^0$, and if a CNF formula u has an instance $u\theta \in L_2$ then $y\theta \in L_2^0$.

[3] Our choice of the approximate specification is rather arbitrary. S_1^0 could be shrunk by additionally requiring in the definition of L_1^0 that $t_j, t_j' \in \{\texttt{true}, \texttt{false}\}$ for $j = 1, \ldots, n$. S_1 can be extended by setting $L_1 = \{\, t \in \mathscr{HU} \mid$ if t is of the form $[t_1\text{-}u_1, \ldots, t_n\text{-}u_n]$ then $t_i = u_i$ for some $i \,\}$ (Drabent 2012). However the chosen S_1^0, S_1 seem more convenient.

Let us also note that in a suitable typed logic the need for an approximate specification would disappear for this particular problem. Roughly speaking, if the argument of *sat_cnf* is restricted to the type of (representations of) CNF formulae, then the set to be defined by *sat_cnf* is unique. The set is specified by the shrunk S_1^0, described above in this footnote.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

By (3.1), a ground *sat_cl*-atom is in S_1 iff it is of the form $A_1 = sat_cl([t\text{-}t|s'])$ or $A_2 = sat_cl([t|s])$ where $s \in L_1$. For the first case, we provide a rule

$$sat_cl([Pol\text{-}Pol|Pairs]). \tag{3.5}$$

Obviously, the rule covers each such A_1, and the sufficient condition for correctness w.r.t. S_1 (Theorem 5) is satisfied (as each ground instance $sat_cl([p\text{-}p|s])$ of (3.5) is in S_1). For the second case, a rule

$$sat_cl([H|Pairs]) \leftarrow sat_cl(Pairs). \tag{3.6}$$

covers each such A_2 (as A_2 is the head of a ground instance $sat_cl([t|s]) \leftarrow sat_cl(s)$ of (3.6), where $sat_cl(s) \in S_1$). The sufficient condition for correctness holds for each ground instance of (3.6), as $sat_cl(s') \in S_1$ implies $s' \in L_1$, hence $[t'|s'] \in L_1$ and $sat_cl([t'|s']) \in S_1$.

A ground *sat_cnf*-atom is in S_1 iff it is $B_1 = [\,]$ or of the form $B_2 = sat_cnf([s|u])$ where $s \in L_1$ and $u \in L_2$. The following rules cover B_1, respectively B_2 w.r.t. S_1, (simple details are left to the reader).

$$sat_cnf([\,]). \tag{3.7}$$

$$sat_cnf([Clause|Clauses]) \leftarrow sat_cl(Clause),\ sat_cnf(Clauses). \tag{3.8}$$

If $sat_cl(s), sat_cnf(u) \in S_1$, then $s \in L_1$, $u \in L_2$, hence, $sat_cnf([s|u]) \in S_1$. So the sufficient condition of Theorem 5 holds for rule (3.8). It obviously holds for rule (3.7). So we constructed program P_1, consisting of rules (3.5)–(3.8), and proved that it is semi-complete and correct w.r.t. S_1.

Termination and completeness of P_1. To show that the program is complete, we show that it is recurrent (cf. Corollary 12). Let us define a level mapping $|\ |: \mathscr{HB} \to \mathbb{N}$ (and an auxiliary mapping $|\ |: \mathscr{HU} \to \mathbb{N}$ for terms):

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

for any ground terms $h, t, t', t_1, \ldots, t_n$, and any function symbol f. Note that $|[t_1,\ldots,t_n]| = 1 + \Sigma_{i=1}^{n} |t_i|$, and that $|t| > 0$ for any term t. It is easy to show that the program P_1 is recurrent under the level mapping $|\ |$, i.e. for each ground instance $H \leftarrow \ldots, B, \ldots$ of a rule of P_1, we have $|H| > |B|$. For example, for a ground instance $sat_cnf([s|u]) \leftarrow sat_cl(s), sat_cnf(u)$ of (3.8), we have $|sat_cnf([s|u])| = |s| + |u|$, which is greater both than $sat_cl(s) = |s|$ and $sat_cnf(u) = |u|$. (We leave further details to the reader.) By Corollary 12, P_1 is complete w.r.t. S_1.

As a side effect, we obtain termination of P_1 for the intended queries (as for any CNF formula t, $|t\theta| = |t|$ for any its instance $t\theta$; thus, query $Q = sat_cnf(t)$ is bounded w.r.t. $|\ |$, and by Theorem 11 P_1 terminates for Q).

Summary. We chose, out of a few possibilities, the specification S_1 of a SAT solver program, and then constructed such a program, namely P_1. The construction was guided by the sufficient condition for semi-completeness and performed hand in hand with a semi-completeness and correctness proof.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Note that the program is not correct w.r.t. S_1^0 (another specification considered here) as, for instance, $sat_cl([a, a\text{–}a]) \notin S_1^0$ is an answer of P_1. The reader is encouraged to construct a program correct and complete w.r.t. S_1^0, in order to see that the program is more complicated (and most likely less efficient), as it contains additional checks (that the argument of *sat_cl* is a list and each its element is of the form $t\text{–}u$).

4 Toward 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 intermediate stage. (An impatient reader may find P_3 on page 17.) The construction is guided by a formal specification, and done together with a correctness and semi-completeness proof. It follows the approach of Section 2.4. We only partially discuss the reasons for particular design decisions in constructing the logic programs and in adding control, as the algorithmic and efficiency issues are outside of the scope of this work. The constructed program is proved to be recurrent, from which its completeness and termination follow. The last two subsections concern occur-check freedom of P_3, and inapplicability of correctness-preserving transformations to our example.

In this section, not only the program, but also the specifications are built incrementally Our proofs will remain valid under such changes of specifications by a rather obvious property:

Lemma 14
Let $S' \subseteq S'' \subseteq \mathscr{HB}$ be specifications and P a program.

If an atom A is covered by P w.r.t. S', then it is covered by P w.r.t. S''.

If P does not contain any predicate symbol occurring in $S'' \setminus S'$, and the sufficient condition for correctness from Theorem 5 holds for P w.r.t. S', then the condition holds for P w.r.t. S''.

Program P_1 is correct and complete w.r.t. S_1. However, as explained in Section 3, it is sufficient that predicate *sat_cnf* is correct w.r.t. S_1 and complete w.r.t. S_1^0. So now our program construction will be guided by an approximate specification. Predicates *sat_cnf* and *sat_cl* are going to be correct w.r.t. S_1 and complete w.r.t. S_1^0.

In what follows, SC1 stands for the sufficient condition for correctness, and SC2 stands for the fact that the considered procedure satisfies the sufficient condition for semi-completeness (cf. Theorem 5 and the comment after Theorem 9). Let SC stand for SC1 with SC2. We leave to the reader a simple check of SC2 for *sat_cnf* and S_1^0 (in Section 3, this was done for S_1).

4.1 Preparing for adding control

Program P_1 performs inefficient search by means of backtracking. We are going to improve the search by changing the control. The intended control cannot be applied directly to P_1; so in this section we transform P_1 to a program P_2. The construction

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

of P_2 has two aspects, formal and informal. Formally, we extend the specification in some steps (to describe new predicates), and apply the augmented specifications to construct the rules of P_2 together with proofs of semi-completeness and correctness. At an informal level, the construction of the specifications (and to a certain extent of the program rules) is guided by the intended control, to be eventually added to the program in Section 6.

Program P_2 includes the rules for *sat_cnf* from P_1, i.e. (3.7), (3.8). It contains a new definition of *sat_cl*, and some new predicates. The new predicates and *sat_cl* would define the same set L_{sat_cl} (or its subset $L_{sat_cl} \setminus \{ [t] \mid t \in \mathscr{HU} \}$), where $L_1^0 \subseteq L_{sat_cl} \subseteq L_1$ (cf. (3.1), (3.4)). However, they would represent elements of L_{sat_cl} in a different way.

We are going to improve the search by delaying certain actions. Invoking *sat_cl*(s) is to be delayed when s is a clause with the two first literals non-ground. (Note that a literal being non-ground means that it has no logical value assigned.) In this way, the technique of watched literals (cf. Gomes et al. (2008)) will be implemented. Unification of a non-ground pair p–v (by applying rule (3.5)) is to be performed only if the argument of *sat_cl* is a unit clause, that is $[p\text{–}v]$. This contributes to implementing unit propagation (cf. Gomes et al. (2008)). Removing a literal from a clause (done in P_1 by rule (3.6)) is to be performed only when the literal is ground. Also, if this literal is true, then further search is unnecessary for this clause.

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 of a predicate. 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 $[Pol\text{–}Var|Pairs]$ is represented as three arguments of *sat_cl3*. More formally, its specifications for respectively correctness and completeness are

$$S_{sat_cl3} = \{ sat_cl3(s, v, p) \mid [p\text{–}v|s] \in L_1 \},$$
$$S^0_{sat_cl3} = \{ sat_cl3(s, v, p) \mid [p\text{–}v|s] \in L_1^0 \}.$$

A new procedure for *sat_cl* is obvious:

$$sat_cl([Pol\text{–}Var|Pairs]) \leftarrow sat_cl3(Pairs, Var, Pol). \tag{4.1}$$

SC are trivially satisfied (w.r.t. specifications $S_{11} = S_1 \cup S_{sat_cl3}$, and $S^0_{11} = S^0_1 \cup S^0_{sat_cl3}$, we leave the simple details to the reader).

Following Howe and King (2012), we will use in the program an auxiliary predicate = defining equality. It will turn out necessary when the final Prolog program is constructed. Its specification (for correctness and completeness) is obvious:

$$S_= = \{ t = t \mid t \in \mathscr{HU} \}.$$

As its definition, rule

$$= (X, X). \tag{4.2}$$

will do (as it covers each atom from $S_=$, and satisfies the sufficient condition for correctness of Theorem 5.).

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

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

$$sat_cl3([\,], Var, Pol) \leftarrow Var = Pol. \tag{4.3}$$

Its ground instance $sat_cl3([\,], p, p) \leftarrow p = p$ covers A w.r.t. $S^0_{11} \cup S_=$. Conversely, each instance of (4.3) with the body atom in $S_{11} \cup S_=$ is of this form, its head is in $S_{11} \cup S_=$, 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 by means of `block` declarations of SICStus (Carlsson and Mildner 2012), 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 $[Pol1\text{-}Var1, Pol2\text{-}Var2 \mid Pairs]$ is represented as the five arguments of *sat_cl5*. The intention is to delay selecting *sat_cl5* until its first or third argument is bound (is not a variable). The specifications for correctness and completeness for *sat_cl5* is as follows. (As soon we will need another predicate with the same semantics, the specifications describe both predicates).

$$S_{sat_cl5} = \left\{ \begin{array}{l} sat_cl5(v_1, p_1, v_2, p_2, s), \\ sat_cl5a(v_1, p_1, v_2, p_2, s) \end{array} \middle| \; [p_1\text{-}v_1, p_2\text{-}v_2|s] \in L_1 \right\},$$

$$S^0_{sat_cl5} = \left\{ \begin{array}{l} sat_cl5(v_1, p_1, v_2, p_2, s), \\ sat_cl5a(v_1, p_1, v_2, p_2, s) \end{array} \middle| \; [p_1\text{-}v_1, p_2\text{-}v_2|s] \in L^0_1 \right\}.$$

Now, the specifications for the whole program P_2 are

$$S_2 = S_1 \cup S_= \cup S_{sat_cl3} \cup S_{sat_cl5} \quad \text{(for correctness), and}$$
$$S^0_2 = S^0_1 \cup S_= \cup S^0_{sat_cl3} \cup S^0_{sat_cl5} \quad \text{(for completeness).}$$

The following rule completes the definition of *sat_cl3*.

$$\begin{array}{l} sat_cl3([Pol2\text{-}Var2|Pairs], Var1, Pol1) \leftarrow \\ \qquad sat_cl5(Var1, Pol1, Var2, Pol2, Pairs). \end{array} \tag{4.4}$$

To check SC, let $S = S_2$, $L = L_1$ or $S = S^0_2$, $L = L^0_1$. Then for each ground instance of (4.4), 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 $[p_1\text{-}v_1, p_2\text{-}v_2|s] \in L$ iff $sat_cl3([p_2\text{-}v_2|s], v_1, p_1) \in S$). Hence, SC1 holds for (4.4), and each $sat_cl3([p_2\text{-}v_2|s], v_1, p_1) \in S^0_2$ is covered by (4.4). Thus, each $sat_cl3(s', v_1, p_1) \in S^0_2$ is covered by (4.3) or (4.4)[4].

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*.

$$sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs). \tag{4.5}$$

$$sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \tag{4.6}$$

[4] Notice that some atoms of the form $sat_cl(s), sat_cl3(s, v, p)$ from $S_2 \setminus S^0_2$ are not covered (e.g. when $s = [a, true\text{-}true]$); this is a 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).

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Note that $[p_1{-}v_1, p_2{-}v_2|s] \in L_1$ iff $[p_2{-}v_2, p_1{-}v_1|s] \in L_1$; the same for L_1^0. It immediately follows that SC are 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 (4.5) or without (4.6) remains semi-complete.) The control will choose the one of them which 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 $[p_1{-}v_1, p_2{-}v_2|s] \in L_1^0$ iff $p_1 = v_1$ or $[p_2{-}v_2|s] \in L_1^0$ iff $p_1 = v_1$ or $sat_cl3(s, v_2, p_2) \in S_2^0$. So two rules follow:

$$sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow Var1 = Pol1. \tag{4.7}$$

$$sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl3(Pairs, Var2, Pol2). \tag{4.8}$$

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 $sat_cl5a(\ldots) \in S_2^0$. To check SC1, consider a ground instance of (4.7), with the body atom in S_2. So it is of the form

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

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

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

of (4.8), 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 $sat_cl5a(v_1, p_1, v_2, p_2, s) \in S_2$.

From a declarative point of view, our program is ready. The logic program P_2 consists of rules (3.7)–(4.8). As it satisfies SC, it is correct w.r.t. S_2 and semi-complete w.r.t. S_2^0.

4.2 Avoiding floundering

As described above, program P_2 is intended to be executed with delaying certain calls of procedure *sat_cl5*. A *sat_cl5*-atom will not be selected when the first and the third argument of *sat_cl5* are both variables (SICStus block declaration `sat_cl5(-,?,-,?,?)`). So the program may flounder; a non-empty query with no selected atom may appear in a computation[5]. Floundering is a kind of pruning of SLD-trees, and may cause incompleteness. We now augment P_2 to avoid floundering. Then we prove that the obtained program P_3 actually does not flounder (under the intended selection rule, for the intended initial queries).

Consider an SLD-derivation starting from a query $Q_0 = sat_cnf(u)$, where u is a CNF formula. Speaking informally, in each query in the derivation each variable is bound to some variable of u. So to avoid floundering, we take care that each variable of Q_0 is eventually bound to a constant. This may be seen as applying the test-and-generate paradigm. We add a top level two-argument predicate *sat*. It

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

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

defines the Cartesian product of two sets: (i) the relation (set) defined by *sat_cnf*, and (ii) the set of lists of truth values (i.e. of `true` or `false`). The intention is to use initial queries of the form

$$sat(u, l), \text{ where } u \text{ is a (representation of a) propositional formula in CNF,} \quad l \text{ is the list of the variables in } u. \tag{4.9}$$

For such queries, each variable of u will be eventually bound to `true` or `false`, thus floundering is avoided.

More formally, the specification for the new predicate, respectively, for its correctness and completeness is

$$S_{sat} = \{\, sat(u,t) \mid u \in L_2,\ t \in TFL \,\},$$
$$S^0_{sat} = \{\, sat(u,t) \mid u \in L^0_2,\ t \in TFL \,\},$$

where

$$TFL = \{\, [t_1, \ldots, t_n] \mid n \geqslant 0,\ t_i = \texttt{true} \text{ or } t_i = \texttt{false} \text{ for } i = 1, \ldots, n \,\}.$$

Now, the approximate specification for the extended program consists of

$$S^0_3 = S^0_2 \cup S^0_{sat} \cup S_{tf}, \qquad S_3 = S_2 \cup S_{sat} \cup S_{tf},$$

where

$$S_{tf} = \{\, tflist(t) \mid t \in TFL \,\} \cup \{\, tf(\texttt{true}), tf(\texttt{false}) \,\}$$

describes two auxiliary predicates. The three new predicates are defined in a rather obvious way, following Howe and King (2012):

$$sat(Clauses, Vars) \leftarrow sat_cnf(Clauses), tflist(Vars). \tag{4.10}$$
$$tflist([\,]). \tag{4.11}$$
$$tflist([Var|Vars]) \leftarrow tflist(Vars), tf(Var). \tag{4.12}$$
$$tf(\texttt{true}). \tag{4.13}$$
$$tf(\texttt{false}). \tag{4.14}$$

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 final logic program P_3 consists of rules (4.10)–(4.14) and of the rules of P_2, i.e. (3.7)–(4.8)[6]. The program is correct w.r.t. S_3 and semi-complete w.r.t. S^0_3.

[6] To present the whole program P_3 in one place, here is P_2. The rules of P_3, except of those marked ▹ here, will appear unchanged in the final Prolog program.

$$sat_cnf([\,]). \tag{3.7}$$
$$sat_cnf([Clause|Clauses]) \leftarrow sat_cl(Clause), sat_cnf(Clauses). \tag{3.8}$$
$$sat_cl([Pol\text{-}Var|Pairs]) \leftarrow sat_cl3(Pairs, Var, Pol). \tag{4.1}$$
$$\triangleright\ =(X, X). \tag{4.2}$$
$$sat_cl3([\,], Var, Pol) \leftarrow Var = Pol. \tag{4.3}$$
$$sat_cl3([Pol2\text{-}Var2|Pairs], Var1, Pol1) \leftarrow sat_cl5(Var1, Pol1, Var2, Pol2, Pairs). \tag{4.4}$$
$$\triangleright\ sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs). \tag{4.5}$$
$$\triangleright\ sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \tag{4.6}$$
$$\triangleright\ sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow Var1 = Pol1. \tag{4.7}$$
$$\triangleright\ sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl3(Pairs, Var2, Pol2). \tag{4.8}$$

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Non-floundering – proof. Analysis tools, like that of Genaim and King (2008), can be used to demonstrate non-floundering of P_3, for initial queries of the form (4.9), under a partial selection rule as described above (King 2012). To make the paper self-contained, we present an explicit proof. We first show that a certain class of programs and queries does not flounder under a certain class of partial selection rules.

Theorem 15 (*non-floundering*)
Consider programs P_1', P_2'. Let F_i be the set of the predicate symbols occurring in P_i' (for $i = 1, 2$), and assume that $F_1 \cap F_2 = \emptyset$. Let us call an atom A an F_i-atom if the predicate symbol of A is in F_i.

Assume that each variable occurring in a rule of P_1' occurs in the head of the rule, and each variable occurring in a rule of P_2' occurs in the body of the rule. Consider a partial selection rule $\mathscr{R}$ under which each floundered query consists solely of non-ground F_1-atoms. Let Q be a query such that each variable of Q occurs in an F_2-atom of Q.

Then no derivation of $P_1' \cup P_2'$ for Q via $\mathscr{R}$ flounders.

We now apply the theorem to our case. Note that the premises of the theorem are satisfied by the program $P_4 = P_3 \setminus \{(4.10)\}$ divided into $P_1' = P_2$ and $P_2' = \{(4.11), \ldots, (4.14)\}$, and by the selection rule $\mathscr{R}$ intended for P_3, and described previously. (Any floundered query consists of non-ground *sat_cl*5-atoms, F_1 contains the predicate symbols of P_2, and $F_2 = \{tflist, tf\}$.) Consider a (finite or infinite) SLD-derivation $Q_0, Q_1, \ldots$ of P_3, where Q_0 is of the form (4.9). Then, $Q_1, \ldots$ is a derivation of P_4, and $Q_1 = sat_cnf(u), tflist(l)$ satisfies the conditions of the theorem. Thus, P_3 does not flounder under $\mathscr{R}$ for the intended initial queries.

Proof (of Theorem 15)
Consider a (finite or infinite) derivation $Q = Q_0, Q_1, \ldots$ of $P_1' \cup P_2'$ via $\mathscr{R}$. We show that (independently from the selection rule)

$$\begin{array}{l}\text{if a variable } X \text{ occurs in an } F_1\text{-atom } A \text{ in } Q_i \ (i > 0)\\ \text{then } X \text{ occurs in an } F_2\text{-atom } B \text{ in } Q_i.\end{array} \tag{4.15}$$

Hence, no Q_i consists of non-ground F_1-atoms, and thus floundering does not occur.

As each variable of a clause of P_1' occurs in the head, we have the following:

$$\begin{array}{l}\text{Assume that in a resolution step an } F_1\text{-atom } A \text{ in a query is replaced}\\ \text{by } (A_1, \ldots, A_k)\theta \text{ (where } \theta \text{ is an mgu of } A \text{ and the head } H \text{ of a rule with}\\ \text{body } A_1, \ldots, A_k).\\ \text{If a variable } X \text{ occurs in } (A_1, \ldots, A_k)\theta, \text{ then } X \text{ occurs in } A\theta.\end{array} \tag{4.16}$$

(As X occurs in some $V\theta$, where V occurs both in the body $A_1, \ldots, A_k$ and in the head H; thus, $V\theta$ occurs both in $H\theta$ and $A\theta$.)

Similarly, due to each variable of a clause of P_2' occurring in the body, we have the following:

$$\begin{array}{l}\text{Assume that } A \text{ is an } F_2\text{-atom and, apart from this, } A, A_1, \ldots, A_k, \theta \text{ are as}\\ \text{in (4.16). If a variable } X \text{ occurs in } A\theta, \text{ then } X \text{ occurs in } (A_1, \ldots, A_k)\theta.\end{array} \tag{4.17}$$

(The explanation is similar.) The proof of (4.15) is by induction. (4.15) holds for Q_0. Assume it holds for a Q_i, $i > 0$. The next query Q_{i+1} is obtained from Q_i by applying

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

an mgu θ after a replacement of an F_1-atom by some F_1-atoms $A_1, \ldots, A_k$, or a replacement of an F_2-atom by some F_2-atoms. Assume that a variable Y occurs in an F_1-atom of Q_{i+1}. Then, either by (4.16) Y occurs in $A\theta$ for the selected F_1-atom A of Q_i, or Y occurs in an atom $A\theta$ of Q_{i+1}, where A is a (not selected) F_1-atom of Q_i. In both cases, Y occurs in $X\theta$, for some X occurring in A; hence, by the inductive assumption, X occurs in some F_2-atom B of Q_i; thus, Y occurs in $B\theta$. Either $B\theta$ is an F_2-atom of Q_{i+1}, or by (4.17) Y occurs in an F_2-atom of Q_{i+1}. □

The author is not aware of any existing method applicable in our case. The sufficient condition for non-floundering of Smaus et al. (1998b, Theorem 3.5), a generalization of that of Apt and Luitjes (1995), is based on assigning types and directions (input or output) to argument positions. For the condition to hold for program P_3, the first and the third argument of *sat_cl5* have to be input. Consequently, some other arguments have to be input, including the first argument of *sat*. But then the condition for initial queries of the form (4.9) is violated (for any type assignment under which the condition holds for P_3). It seems, however, that the condition may be used to construct a proof similar to ours (p. 18), by applying it to query Q_1 and program $P_1' \cup P_2'$.

4.3 Completeness and termination of P_3

We showed that P_3 is correct and semi-complete. To establish its completeness, we show that it is recurrent. Consider a level mapping

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

where t, u, u_1, u_2, u_3, u_4 are arbitrary ground terms, $|t|$ is as in Section 3 and *listsize* is defined by $listsize([h|t]) = listsize(t) + 1$ and $listsize(f(t_1, \ldots, t_n)) = 0$ for any f which is not [|]. For example, consider (4.1). For any its ground instance $sat_cl([p\text{-}v|t]) \leftarrow sat_cl3(t, v, p)$, 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. As it is semi-complete, it is complete by Corollary 12.

As an additional corollary, we obtain a non-declarative property – 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 $s\text{-}s'$, and t' is a list. Each intended query to the program is of this form. Q is bounded (for each its ground instance $Q\theta$, $|Q\theta| = |Q|$). As P_3 is recurrent, each SLD-tree for P_3 and Q is finite.

4.4 Occur-check

The implementation of unification in Prolog is unsound, due to lack of the occur-check. As a result, Prolog may produce answers which, according to the theory, are not answers of the program. Here, we show that the occur-check in the unification algorithm is unnecessary for program P_3, initial queries of the form (4.9), and any

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

selection rule. So the correctness of P_3 w.r.t. S_3 is preserved when the program (with such queries) is executed by Prolog.

First, each rule head of P_3, except for that of (4.2), is linear. So the occur-check is unnecessary to correctly unify such a head with any atom. More formally, the corresponding equation is NSTO (not subject to occur-check) (Apt 1997, Lemma 7.5).

It remains to consider rule (4.2), which is $=(X, X)$. Let $Q_0, Q_1, \ldots$ be an SLD-derivation for P_3 starting from a query Q_0 of the form (4.9). By simple induction on i, we obtain the following facts. In each subterm $t{-}v$ of any query Q_i, t is a constant (it is `true` or `false`). Also, in each *sat_cl*3-atom in Q_i, the third argument is a constant and the second and the fourth arguments of any *sat_cl*5- or *sat_cl*5*a*-atom are constants.

Note now that each $=$-atom $v{=}p$ appearing in a query has been introduced as the result of a resolution step employing rule (4.3) or (4.7). Hence, p is a constant, and the occur-check in resolving $v{=}p$ with rule (4.2) is unnecessary – the Martelli–Montanari unification algorithm eventually unifies v with p[7]. To summarize, P_3 can be safely executed under the unsound unification of Prolog, lacking the occur-check.

A comment is due that the notion of a NSTO program (with a query) of (Apt 1997) is irrelevant here, as it guarantees occur-check freedom only for LD-derivations. The corresponding notion of Deransart and Małuszyński (1993) deals with arbitrary selection rules, but P_3 with Q_0 does not satisfy the first two sufficient conditions for NSTO given in that work (Proposition 8.3, Theorem 8.5). The third sufficient condition involves a substantial transformation of the program, and seems substantially more complicated than the proof presented here. Also, the method of Apt and Luitjes (1995) and its generalizations of Smaus et al. (1998b, 1998a) are inapplicable in our case. The methods are based on modes – classifying argument positions as input or output. Occur-check freedom is guaranteed for so called nicely moded programs and queries. In our case, P_3 with Q_0 is not nicely moded under any moding[8].

4.5 On correctness-preserving transformations

Here, we discuss inapplicability of the paradigm of semantics preserving program transformations to our example.

Let us first point out that our example shows usefulness of approximate specifications. In Section 3, we showed that the problem statement naturally leads to an approximate specification. It is unnecessary to know the exact relations defined by the program; what matters is its full correctness w.r.t. the approximate specification (i.e. correctness w.r.t. the specification for correctness and completeness w.r.t. that for

[7] Formally, this follows from analysis of the behaviour of the algorithm. Another proof can be obtained by Lemma 7.5 of Apt (1997).

Also, having proved that p is ground in any $v{=}p$ in any query, we obtain the occur-check freedom by Occur-check Freedom 1 theorem of Apt and Luitjes (1995) (with the second position of $=$ treated as input, and all the other argument positions as output).

[8] At least one position of $=$ must be output; hence, some positions of *sat_cl*3, *sat_cl*5*a*, *sat_cl*5 and *sat_cl* must be output. Thus, the first position of *sat_cnf* is output; this requires the (term representing a) clause in the initial query to be linear.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

completeness). Note that the exact semantics of programs P_2 and P_3 is not obvious, and describing it may be rather cumbersome. For instance, *sat_cl* defines (both in P_2 and P_3) the set of terms of the form $[t\text{-}t]$, or $[t_1\text{-}u_1, \ldots, t_n\text{-}u_n|s]$ $(n > 1)$, where $t_i = u_i$ for some i. Moreover, the semantics of (the common predicates in) the consecutive versions P_1, P_2 of the program differ. For instance, $A = sat_cl([\texttt{true-true}|1]) \in \mathscr{M}_{P_1}$, but $A \notin \mathscr{M}_{P_2}$. What does not change, is the correctness and completeness w.r.t. a fixed approximate specification (of the common predicates of P_1, P_2, P_3).

This shows a weakness of the paradigm of program development by semantics-preserving transformations (Pettorossi et al. 2010, and the references therein). Constructing P_3 by semantics-preserving transformations seems completely unfeasible. The exact semantics of (the common predicates in) the final program depends on particular design decisions made at the development steps and cannot be known in advance. Thus, our example suggests that a more general paradigm of program transformations is worth introducing – transformations which preserve correctness and completeness w.r.t. an approximate specification.

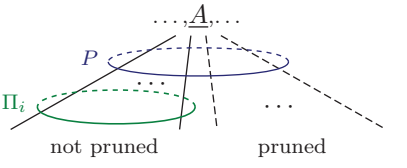
5 Pruning and completeness

In the previous section, we constructed a logic program P_3, which is correct and complete w.r.t. the respective specifications and, for the intended queries, terminates, is occur-check free, and does not flounder under the intended selection rule. The final Prolog program will be obtained from P_3 by adding control; this includes pruning. This section presents a way of proving that pruning does not violate program completeness.

Pruning means removing some parts of SLD-trees. In Prolog, pruning can be performed by means of the cut, or devices like `once/1` or the if-then-else construct. Pruning preserves correctness of a program; it also preserves termination, occur-check freedom and non-floundering. However, it may violate program completeness. We present a method (Drabent 2016a) of showing that completeness is actually preserved. The presentation follows Drabent (2016a). This method is based on a rather abstract view of pruning; a (more complicated) approach directly referring to the cut is presented in Drabent (2017).

5.1 Pruned SLD-trees

We will view pruning of an SLD-tree as applying only certain rules while constructing the children of a node. To formalize this, we introduce subsets $\Pi_1, \ldots, \Pi_n$ of a given program P. The intention is that for each node the rules of one Π_i are used. The diagram (Drabent 2015) compares selection of atom A in an SLD-tree with selection of A and Π_i in a pruned tree.



https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Definition 16

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

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

A csSLD-tree T with root Q is **complete** w.r.t. a specification S if, for any ground $Q\theta$, $S \models Q\theta$ implies that $Q\theta$ is an instance of an answer of T.

Informally, such a complete tree produces all the answers for Q required by S. A notion similar to c-selection rule was used by Pedreschi et al. (2002).

In Prolog with delays, it is possible that no atom is selected in a non-empty query. This can be modelled as a c-selection rule which for such query selects program $\emptyset$.

5.2 Reasoning about completeness of csSLD-trees

Consider programs $P, \Pi_1, \ldots, \Pi_n$ and specifications $S, S_1, \ldots, S_n$, such that $P \supseteq \bigcup_{i=1}^n \Pi_i$ and $S = \bigcup_{i=1}^n S_i$. The intention is that each S_i describes which answers are to be produced by using Π_i in the first resolution step. We will call $\Pi_1, \ldots, \Pi_n, S_1, \ldots, S_n$ a **split** (of P and S).

Definition 17

Let $\mathcal{S} = \Pi_1, \ldots, \Pi_n, S_1, \ldots, S_n$ be a split, and $S = \bigcup S_i$. Specification S_i is **suitable** for an atom A w.r.t. $\mathcal{S}$ when $ground(A) \cap S \subseteq S_i$. (In other words, when no instance of A is in $S \setminus S_i$.) We also say that a program Π_i is **suitable** for A w.r.t. $\mathcal{S}$ when S_i is.

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

- $\Pi \in \{\Pi_1, \ldots, \Pi_n\}$ is suitable for A w.r.t. $\mathcal{S}$, or
- none of $\Pi_1, \ldots, \Pi_n$ is suitable for A w.r.t. $\mathcal{S}$ and $\Pi = \emptyset$ (so Q' is a leaf).

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

The intuition is that when Π_i is suitable for A, then S_i is a fragment of S sufficient to deal with A. It describes all the answers for query A required by S. For instance, consider a specification $S = \{ p(t) \mid t \in L_1 \}$ (cf. (3.1)) and a split $\mathcal{S} = \Pi_1, \Pi_2, S_1', S_2'$, where $S_1' = \{ p([t\text{-}t|s]) \mid t, s \in \mathscr{HU} \}$ and $S_2' = S$ (we skip the details of Π_1, Π_2, note that $S_1' \subseteq S$). Then, S_1' (and Π_1) is suitable for an atom $A = p([X\text{-}X|T])$, and is not suitable for $p([X\text{-}Y|T])$.

In a compatible tree, Π_i is selected together with A. If now each atom of S_i is covered by Π_i w.r.t. S, then using for A only the rules of Π_i does not destroy completeness. Formally (Drabent 2016a):

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

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

(1) for each $i = 1, \ldots, n$, all the atoms from S_i are covered by Π_i w.r.t. S, and
(2) T is compatible with $\Pi_1, \ldots, \Pi_n, S_1, \ldots, S_n$,
(3) T is finite or P is recurrent

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

6 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[9]. The control added to P_3 modifies the default Prolog selection rule (by means of delays), and prunes some redundant parts of the search space (by the if-then-else construct). So correctness, termination and occur-check freedom of P_3 are preserved (as the latter two properties have been proved for any selection rule).

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

```
:- block sat_cl5(-, ?, -, ?, ?).
```
(6.1)

For the intended initial queries, such delaying does not lead to floundering (as shown in Section 4.2). So the completeness of the logic program is preserved.

The first case of pruning is to use only one of the two rules (4.5), (4.6),

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

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:

$$\begin{aligned} sat_cl5(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow \\ nonvar(Var1) \rightarrow sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \\ ; \; sat_cl5a(Var2, Pol2, Var1, Pol1, Pairs). \end{aligned} \tag{6.2}$$

The other case of pruning concerns rules (4.7), (4.8):

$$sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow Var1 = Pol1.$$
$$sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow sat_cl3(Pairs, Var2, Pol2).$$

Rule (4.8) is to be skipped when (4.7) succeeds. (Here, we prune further search when clause $[Pol1, Var1, Pol2, Var2 \,|\, Pairs]$ is found true.) This is done by converting the two rules into

$$\begin{aligned} sat_cl5a(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow \\ Var1 = Pol1 \; \rightarrow \; true \; ; \; sat_cl3(Pairs, Var2, Pol2). \end{aligned} \tag{6.3}$$

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

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

This completes our construction. The obtained Prolog program consists of declaration (6.1), the rules of P_3 except for those for *sat_cl5*, *sat_cl5a* and =, i.e. (3.7)–(4.1), (4.3), (4.4), (4.10)–(4.14), and Prolog rules (6.2), (6.3). Rule (4.2) for = has to be skipped, as it is built in Prolog.[10]

Completeness of the final program. We now prove that completeness is preserved under pruning imposed on P_3 by the control described above. Let us construct a split $\mathcal{S} = \Pi_0, \ldots, \Pi_5, S'_0, \ldots, S'_5$ of program P_3 and its specification for completeness S^0_3:

$$\begin{array}{ll}
\Pi_1 = \{(4.5)\}, & S'_1 = S'_2 = S^0_3 \cap \{\, sat_cl5(t_1, \ldots, t_5) \mid t_1, \ldots, t_5 \in \mathscr{HU} \,\} \\
\Pi_2 = \{(4.6)\}, & \quad = \{\, sat_cl5(v_1, p_1, v_2, p_2, s) \mid [p_1{-}v_1, p_2{-}v_2|s] \in L^0_1 \,\}, \\
\Pi_3 = \{(4.7)\}, & S'_3 = \{\, sat_cl5a(p, p, v_2, p_2, s) \mid [p{-}p, p_2{-}v_2|s] \in L^0_1 \,\}, \\
\Pi_4 = \{(4.8)\}, & S'_4 = \{\, sat_cl5a(v_1, p_1, v_2, p_2, s) \mid v_1 \neq p_1,\ [p_2{-}v_2|s] \in L^0_1 \,\}, \\
\Pi_5 = P_3 \setminus \bigcup_{i=1}^4 \Pi_i, & S'_5 = S^0_3 \setminus (S'_1 \cup S'_3 \cup S'_4).
\end{array}$$

Note that S'_5 does not contain any *sat_cl5*-atoms or *sat_cl5a*-atoms, and that $\bigcup_{i=1}^5 S'_i = S^0_3$. In particular, $S'_3 \cup S'_4$ is the set of all *sat_cl5a*-atoms from S^0_3. To apply Theorem 18, we need to show that for $i = 1, \ldots, 5$ each atom of S'_i is covered by Π_i w.r.t. S^0_3. A proof of this fact has already been done; it consists of the relevant fragments of the proof that each atom S^0_3 is covered by P_3, spread over Sections 3, 4.

In what follows, let us consider a pruned tree T with the root of the form (4.9). It can be seen as a csSLD-tree for $\Pi_1, \ldots, \Pi_5$. The c-selection rule selects Π_1 or Π_2 for *sat_cl5*-atoms, Π_3 for *sat_cl5a*-atoms with the first two arguments unifiable, Π_4 for the remaining *sat_cl5a*-atoms and Π_5 for the atoms with the other predicate symbols.

We now show that the tree T is compatible with $\mathcal{S}$. When Π_i, where $i \in \{1, 2, 5\}$, is selected for a q-atom A then the corresponding specification S'_i contains all the q-atoms from S^0_3. So $ground(A) \cap S^0_3 \subseteq S'_i$. It remains to consider A being a *sat_cl5a*-atom. Due to the delays implemented by (6.1) and pruning implemented in (6.2), when a *sat_cl5a*-atom A is selected in a node of the tree then its first argument is not a variable, say it is $f(t_1, \ldots, t_n)$ ($n \geqslant 0$). In Section 4.4, we showed that the second argument is a constant c (which is `true` or `false`). So $A = sat_cl5a(f(t_1, \ldots, t_n), c, \ldots)$. Thus, either $ground(A) \cap S^0_3 \subseteq S'_3$ (when the first two arguments in A are unifiable, hence are equal) and Π_3 is selected for A, or $ground(A) \cap S^0_3 \subseteq S'_4$ (otherwise) and Π_4 is selected for A. Remember that in Section 4.2 we showed that in each non-empty query of the tree some atom is selected (there is no floundering). Thus, T is compatible with $\mathcal{S}$.

Remember also that P_3 is recurrent. So the premises of Theorem 18 are satisfied, and each pruned tree with the root of the form (4.9) is complete w.r.t. S^0_3.

Remarks. The Prolog program constructed in this section is correct w.r.t. specification S_3 and, for initial queries of the form (4.9), it terminates under any selection rule,

[10] Note that whenever $v = p$ originating from (6.3) is selected (in a computation starting from an initial query of the form (4.9)) then v is not a variable (by (6.2) and (6.1)) and p is a constant (as shown in Section 4.4). Thus, = in (6.3) can be replaced by a non-logical built-in == of Prolog, as it is done in Howe and King (2012).

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

does not flounder, and is occur-check free. These properties follow from those of logic program P_3, proved in Section 4. Here we proved that, for queries of this form, the Prolog program preserves the completeness of P_3 w.r.t. S_3^0. In practice, this means that the program will produce all the answers required by the specification. Thus, by (3.3), the program will find all the solutions to the SAT problem represented by the query.

7 Conclusions

The example. This paper presents a construction of a non-toy Prolog program, together with proofs of its correctness, completeness, termination, occur-check freedom and non-floundering. The program is the SAT solver of Howe and King (2012). It is seen as a definite clause logic program with added control. Starting from a formal specification, a sequence of definite clause programs, called P_1, P_2, P_3, is constructed hand in hand with proofs of their correctness and completeness (Sections 3 and 4). The construction is guided by the proofs, mainly those of semi-completeness. This suggests a general approach (Section 2.4) to constructing programs which are provably correct and semi-complete. The author believes that the approach, possibly treated informally, should be useful in practice.

Additionally, for the assumed class of initial queries program P_3 is proved to terminate, not flounder, and be occur-check free; termination and occur-check freedom hold for arbitrary selection rules, and non-floundering – for a class of selection rules including that to be used in the final Prolog program.

The Prolog program is obtained from P_3 by adding control – imposing a specific selection rule and pruning the SLD-trees. The pruning is done by means of the if-then-else construct (and can equivalently be done by means of the cut). Such control preserves correctness; it also preserves occur-check freedom, and termination (as in this particular case, they are independent from the selection rule). However, it may violate completeness due to pruning and floundering. Non-floundering has already been proven for P_3. In Section 6, we prove that the completeness is preserved under pruning.

Proof methods. We applied general methods to prove correctness (Clark 1979), completeness (Drabent 2016a), termination (Bezem 1993) and completeness under pruning (Drabent 2016a). The known methods for proving non-floundering and occur-check freedom seem inapplicable in our case (cf. Sections 4.4, 4.2). To deal with non-floundering, we introduced a sufficient condition, suitable for a certain class of programs. The proof of occur-check freedom is rather *ad hoc*. It should be added that non-floundering and termination can be determined by means of automatic tools, see Genaim and King (2008) for the former, and e.g. Nguyen et al. (2011, and the references therein) for the latter.

The employed proof methods for correctness and completeness were described and discussed by Drabent (2016a). The method for program correctness (Clark 1979) is simple, natural, should be well-known, but is often neglected. Instead, more complicated methods are proposed, like that used by Apt (1997). Moreover, these

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

methods are not declarative (cf. footnote 1, see Drabent and Miłkowska (2005) and Drabent (2016a) for further discussion). The methods for program completeness, and for completeness of pruned SLD-trees are due to Drabent (2016a). The approach for completeness introduces a notion of semi-completeness. Roughly speaking, semi-completeness and termination imply completeness. Semi-completeness alone may be useful – if a semi-complete program terminates, then all the answers required by the specification have been obtained. The presented sufficient condition for semi-completeness is a necessary condition for program completeness (in a sense made precise in Drabent (2016a)); so semi-completeness may be understood as a necessary step when dealing with completeness.

A subject for future work is formalization and automation of the presented proof methods. This requires introducing a particular language to express (a chosen class of) specifications. The next issue is devising formal methods of checking the sufficient conditions used here.

In the author's opinion, the sufficient conditions for correctness and semi-completeness are simple and natural. For correctness, one has to show that the rules of the program produce specified atoms (cf. Definition 1) out of specified ones. For semi-completeness – that each specified atom can be produced by some rule of the program out of some specified atoms. The author believes that the two proof methods and the program construction approach of Section 2.4 are a practical and useful tool. They should be applicable, possibly at a less formal level, in actual everyday programming, and in teaching logic programming. The program development case of Sections 3 and 4, maybe dealt with less formally, can be considered a practical example. Conversely, the two proof methods can be understood as a formalization of a usual way in which competent programmers reason declaratively about their logic programs.

Declarativeness. Logic programming could not be considered a declarative programming paradigm unless there exist declarative ways of reasoning about program correctness and completeness. (We remind that by declarative we mean referring only to logical reading of programs, thus abstracting from any operational semantics.) The methods for proving correctness and semi-completeness are purely declarative. So is the completeness proving method used in this paper; however, it is applicable to a restricted class of programs.

In general, completeness proofs based on semi-completeness may be not declarative, as they refer, maybe indirectly, to program termination. So we have a compromise between declarative and non-declarative reasoning: declarative proofs of correctness and semi-completeness, and a non-declarative step from semi-completeness to completeness. Nevertheless, such compromise should be useful in the practice of declarative programming, as usually termination has to be established anyway. This is also supported by the fact that a declarative method for proving completeness (Deransart and Małuszyński 1993) results in a proof similar to a proof of semi-completeness together with a proof of termination (Section 2.3); however, such proof does not imply termination.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Approximate specifications. We point out usefulness of approximate specifications. They are crucial for avoiding unnecessary complications in constructing specifications and in correctness and completeness proofs. They are natural, as when starting construction of a program, the relations it should compute are often known only approximately. Sections 3 and 4 provide an example. Also, it is often cumbersome (and unnecessary) to exactly establish the relations computed by a program. This is the case in our example, as discussed in Section 4.5. See Drabent (2016a) for further examples and discussion.

Program transformations. The program development example presented here differs from those employing semantics-preserving transformations (Pettorossi et al. 2010, and the references therein). In our example, the semantics of (the common predicates in) the consecutive versions of the program differ. What is unchanged, is the correctness and completeness w.r.t. an approximate specification. In Section 4.5, we argue that the paradigm of semantics-preserving transformations is inapplicable to our case. This suggests introducing a more general paradigm of program transformations which preserve correctness and completeness w.r.t. an approximate specification.

Final remarks. We are interested in declarative programming. The example presented here is intended to show how much of the programming task can be done declaratively, 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. At this stage, correctness, completeness, termination, non-floundering and occur-check freedom were formally proven; the reasoning about correctness and completeness was declarative. It is important that all the considerations and decisions about the program execution and efficiency (only superficially treated in this paper) are independent from those related to the declarative semantics, in particular to the correctness and completeness of the program.

We argue that the employed proof methods and the program construction approach 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 (in addition to rather small examples in the previous work) by the main example of this article.

Acknowledgement

Extensive comments of anonymous referees were helpful in improving the presentation.

References

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

Apt, K. R. and Luitjes, I. 1995. Verification of logic programs with delay declarations. In *Proc. of Algebraic Methodology and Software Technology, AMAST '95*, V. S. Alagar and M. Nivat, Eds. Lecture Notes in Computer Science, vol. 936. Springer, 66–90.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Apt, K. R. and Pedreschi, D. 1993. Reasoning about termination of pure Prolog programs. *Information and Computation 106,* 1, 109–157.

Bezem, M. 1993. Strong termination of logic programs. *Journal of Logic Programming 15,* 1&2, 79–97.

Carlsson, M. and Mildner, P. 2012. SICStus Prolog – the first 25 years. *TPLP 12,* 1–2, 35–66.

Clark, K. L. 1979. *Predicate Logic as Computational Formalism.* Technical Report 79/59, Imperial College, London. December.

Davis, M., Logemann, G. and Loveland, D. W. 1962. A machine program for theorem-proving. *Communications of the ACM 5,* 7, 394–397.

Deransart, P. 1993. Proof methods of declarative properties of definite programs. *Theoretical Computer Science 118,* 2, 99–166.

Deransart, P. and Małuszyński, J. 1993. *A Grammatical View of Logic Programming*. The MIT Press.

Deville, Y. 1990. *Logic Programming: Systematic Program Development*. Addison-Wesley.

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), Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany, vol. 17, 301–311. http://drops.dagstuhl.de/opus/volltexte/2012/3631.

Drabent, W. 2015. On completeness of logic programs. In *Logic Based Program Synthesis and Transformation, LOPSTR 2014. Revised Selected Papers*. Lecture Notes in Computer Science, vol. 8981. Springer. Extended version in *CoRR* abs/1411.3015 (2014). http://arxiv.org/abs/1411.3015.

Drabent, W. 2016a. Correctness and completeness of logic programs. *ACM Transactions on Computational Logic 17,* 3, 18:1–18:32.

Drabent, W. 2016b. On definite program answers and least Herbrand models. *TPLP 16,* 4, 498–508.

Drabent, W. 2017. Proving completeness of logic programs with the cut. *Formal Aspects of Computing 29,* 1, 155–172.

Drabent, W. and Miłkowska, M. 2005. Proving correctness and completeness of normal programs – a declarative approach. *TPLP 5,* 6, 669–711.

Genaim, S. and King, A. 2008. Inferring non-suspension conditions for logic programs with dynamic scheduling. *ACM Transactions on Computational Logic 9,* 3, 17:1–17:43.

Gomes, C. P., Kautz, H., Sabharwal, A. and Selman, B. 2008. Satisfiability solvers. In *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz and B. Porter, Eds. Chapter 2, Elsevier, 89–134.

Howe, J. M. and King, A. 2012. A pearl on SAT and SMT solving in Prolog. *Theoretical Computer Science 435*, 43–55.

King, A. 2012. Private communication.

Kowalski, R. A. 1979. Algorithm = logic + control. *Communications of the ACM 22,* 7, 424–436.

Maher, M. J. 1988. Equivalences of logic programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, 627–658.

Nguyen, M. T., Schreye, D. D., Giesl, J. and Schneider-Kamp, P. 2011. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *TPLP 11,* 1, 33–63.

Pedreschi, D. and Ruggieri, S. 1999. Verification of logic programs. *Journal of Logic Programming 39,* 1–3, 125–176.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press

Pedreschi, D., Ruggieri, S. and Smaus, J.-G. 2002. Classes of terminating logic programs. *TPLP 2,* 3, 369–418.

Pettorossi, A., Proietti, M. and Senni, V. 2010. The transformational approach to program development. In *A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming, GULP*, A. Dovier and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 6125. Springer, 112–135.

Pfenning, F., Ed. 1992. *Types in Logic Programming*. The MIT Press.

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

Smaus, J., Hill, P. M. and King, A. 1998a. Preventing instantiation errors and loops for logic programs with multiple modes using block declarations. In *Proc. of Logic Programming Synthesis and Transformation, LOPSTR'98*, P. Flener, Ed. Lecture Notes in Computer Science, vol. 1559. Springer, 289–307.

Smaus, J., Hill, P. M. and King, A. 1998b. Termination of logic programs with block declarations running in several modes. In *Proc. of Principles of Declarative Programming, PLILP'98*, C. Palamidessi, H. Glaser and K. Meinke, Eds. Lecture Notes in Computer Science, vol. 1490, 73–88.

https://doi.org/10.1017/S1471068417000047 Published online by Cambridge University Press