TPLP **25** (4): 631–648, 2025. © The Author(s), 2025. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068425100239 First published online 12 September 2025

On Systematic Construction of Correct Logic Programs

WŁODZIMIERZ DRABENT
Institute of Computer Science, Polish Academy of Sciences, Poland
(*e-mail:* drabent@ipipan.waw.pl)

submitted 27 July 2025; revised 27 July 2025; accepted 29 July 2025

Abstract

Partial correctness of imperative or functional programming divides in logic programming into two notions. Correctness means that all answers of the program are compatible with the specification. Completeness means that the program produces all the answers required by the specifications. We also consider semi-completeness – completeness for those queries for which the program does not diverge. This paper presents an approach to systematically construct provably correct and semi-complete logic programs, for a given specification. Normal programs are considered, under Kunen's 3-valued completion semantics (of negation as finite failure) and the well-founded semantics (of negation as possibly infinite failure). The approach is declarative, it abstracts from details of operational semantics, like, for example, the form of the selected literals ("procedure calls") during the computation. The proposed method is simple and can be used (maybe informally) in actual everyday programming.

KEYWORDS: negation in logic programming, program correctness, program completeness, specifications, program synthesis, teaching logic programming

This paper presents a method of constructing provably correct and semi-complete normal logic programs. The considered semantics are Kunen semantics (3-valued completion semantics, corresponding to negation as finite failure, NAFF), and the well-founded semantics (corresponding to negation as, possibly infinite, failure). The approach is declarative, it abstracts from details of operational semantics, like, for example, the form of the selected literals ("procedure calls") during the computation. The approach can be used, possibly at an informal level, in programmers' practice. It can be seen as a guidance on how to construct programs and thus used in teaching logic programming.

In this paper, we first recall the method for definite clause programs, presented earlier. Then "Preliminaries" present the main concepts, including both semantics of interest. The next section discusses specifications for programs with negation. Then, for each semantics, we present ways of proving correctness and completeness, and a method

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

of constructing programs. We assume that the reader is familiar with basics of logic programming, negation in logic programming, and Prolog.

1 Constructing definite clause programs

Here we briefly present a method of Drabent (2018). For this we first informally describe the main concepts related to program correctness. For further details and references see the papers by Drabent (2016, 2018) and Apt (1997). Definite clause programs will be called shortly *positive programs*.

In imperative (or functional) programming, a central notion is partial correctness. A program is partially correct if its termination implies its compatibility with the specification. In logic programming this divides into two notions: correctness and completeness. Correctness of a program P w.r.t. (with respect to) a specification S means that each answer of P is compatible with S, completeness – that each answer required by S is an answer of P. More technically: A specification is an Herbrand interpretation. A program P with the least Herbrand model $\mathcal{M}_P$ is *correct* w.r.t. a specification S if $\mathcal{M}_P \subseteq S$. It is *complete* w.r.t. S if $S \subseteq \mathcal{M}_P$.

By an answer of a positive program P we mean any query Q such that $P \models Q$. (Answer is sometimes called correct, or computed, instance of a query (Apt 1997, Definition 4.6, 3.6).) In practice, if Prolog succeeds for P and a query Q_0 with a substitution θ then $Q_0\theta$ is an answer for P. For a program P correct w.r.t. S, each answer Q of P is compatible with S (i.e. $S \models Q$); for P complete w.r.t. S, each ground answer Q required by S (i.e. $S \models Q$) is an answer of P.

Proving correctness. There exists a simple (however often forgotten) way of proving correctness of positive programs: A sufficient condition for P being correct w.r.t. S is $S \models P$ (Clark 1979). (In other words, the sufficient condition is that for each ground instance of a clause of P, if the body atoms are in S then the head is in S.) Informally: any clause applied to correct atoms produces a correct atom.

Proving completeness. We need some auxiliary notions. A ground atom A is *covered* (Shapiro 1983) w.r.t. S by a clause C, if C has a ground instance $A \leftarrow B_1, \ldots, B_n$ and $B_1, \ldots, B_n \in S$ (informally: if A can be produced by C from S). Given a specification S, program P is *complete for a query* Q if, speaking informally, it produces all the answers for Q required by S. P is *semi-complete* if it is complete for any query for which there exists a finite SLD-tree. To reason about completeness, it is convenient to deal with semi-completeness and termination separately. Often a proof of completeness is similar to a proof of semi-completeness together with a proof of termination; and the termination has to be established anyway.

Again, there is a simple sufficient condition for semi-completeness: If each atom from S is covered w.r.t. S by a clause of P then P is semi-complete w.r.t. S (Drabent 2016). Informally: any atom required to be produced can be produced by a clause of the program out of atoms required to be produced.

It is surprising that reasoning about completeness was often neglected. For example, an important monograph by Apt (1997) does not even mention the notion of program

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

completeness. Completeness for positive programs was dealt with by Deransart and Małuszyński (1993), Drabent and Miłkowska (2005), Drabent (2016), and for normal programs by Malfon (1994) and Drabent and Miłkowska (2005). Sterling and Shapiro (1994) discuss completeness informally.

Approximate specifications. An important observation is that many issues can be simplified if we consider separate specifications for correctness and completeness (Drabent and Miłkowska 2005; Drabent 2016, 2018). A single specification for both has to exactly describe the least Herbrand model of a program. This is often too troublesome (and not necessary). For instance, it is unimportant how *append*/3 of Prolog behaves on non list arguments, or what happens when the second argument of *member*/2 is not a list, or how *insert*/3 of insertion sort inserts a number into a not sorted list. Moreover, in program construction we should not fix in advance the behaviour of the future program in such unimportant cases. (In the main example of the paper by Drabent (2018), such behaviour is different in consecutive versions of the program; otherwise an efficient version could not have been obtained.) A pair (S_{compl}, S_{corr}) of specifications for completeness and correctness, where $S_{compl} \subseteq S_{corr}$, will be called *approximate specification*. A program with the least Herbrand model $\mathcal{M}_P$ will be called *fully correct* w.r.t. (S_{compl}, S_{corr}) if $S_{compl} \subseteq \mathcal{M}_P \subseteq S_{corr}$.

Example 1.
A list membership predicate $m/2$ should define the set $r = \{\, m(e_i, [e_1, \ldots, e_n]) \in \mathcal{HB} \mid 1 \leq i \leq n \,\}$ ($\mathcal{HB}$ is the Herbrand base). However defining exactly r would be inefficient, and the *member*/2 of Prolog defines a superset of r. It is not important what exactly the defined set is. What matters is that whenever the second argument is a list, the first argument is its member. Hence, any ground answer should be a member of $r_+ = \{\, m(e,u) \in \mathcal{HB} \mid \text{if } u \text{ is a list then } m(e,u) \in r \,\} = r \cup \{\, m(e,u) \in \mathcal{HB} \mid u \text{ is not a list} \,\}$. This leads to an approximate specification (r, r_+). It states that the atoms from r (resp. r_+) must (may) be answers of the program.

Program construction. The above sufficient conditions for semi-completeness and correctness suggest a method (Drabent 2018) of constructing a program, given a specification (S_{compl}, S_{corr}). Due to the condition for semi-completeness, we need (A) to construct clauses $C_1, \ldots, C_n$ so that each atom from S_{compl} is covered w.r.t. S_{compl} by some of them. For correctness we need that (B) $S_{corr} \models C_i$ for each such C_i. This leads to a provably correct and semi-complete program. We also need that (C) the clauses are chosen so that the program terminates for queries of interest.

Termination is kept outside of the proposed rigorous method. One may use any way of proving termination; when applying the method informally, one uses the usual hints of Prolog programming craft, like using in the body proper subterms of the terms in the head.

Example 2.
Take specification (r, r_+) from Example 1. For (A) consider an atom $A \in r$. We have two cases. 1. $A = m(e, [e|t])$. Such A is covered by a fact $C_1 = m(E, [E|_])$; note that (B) holds. 2. $A = m(e, [e'|t])$, and e is a member of the list t. This suggests a clause $C_2 = m(E, [_|T]) \leftarrow m(E, T)$. It covers A, and (B) holds. As T is a proper subterm of

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

$[_|T]$, program $M = \{C_1, C_2\}$ terminates for any ground query. Thus M is correct and complete w.r.t. (r, r_+). Example wrong choices in step 2. are: a fact $m(E, [F|T])$ (covers A, but (B) does not hold), a clause $m(E, [_|T]) \leftarrow m(E, [_|T])$ (fulfills (A) and (B), but leads to non-termination). Note that we constructed the built-in procedure member/2 of Prolog.

Example 3.
For a bigger and detailed example, consider finding the middle element of a list, without using numbers in the program. The specification is (S_{mid}, S_{mid}) where $S_{mid} = \{\, mid(e_i, [e_1, \ldots, e_{2i-1}]) \in \mathcal{HB} \mid i > 0 \,\}$. We follow a standard hint – solve a more general problem. Instead of a list of a certain length containing e at a certain position, let us consider a term with e at this position, and a list of this length. This leads to a specification of a new predicate: (S_{mi}, S_{mi}), where $S_{mi} = \{\, mi(e_i, [e_1, \ldots, e_i|t], [f_1, \ldots, f_{2i-1}]) \in \mathcal{HB} \mid i > 0 \,\}$, and specification (S, S), where $S = S_{mid} \cup S_{mi}$, for the program. For (A) we have three cases:

1. $A = mid(e_i, l)$, where $l = [e_1, \ldots, e_{2i-1}]$. Obviously $mi(e_i, l, l) \in S$. So A is covered by $C_1 = mid(E, L) \leftarrow mi(E, L, L)$. To show (B), note that if $mi(e, l', l') \in S$ then $l' = [f_1, \ldots, f_{2i-1}]$ and $e = f_i$, hence $mid(e, l') \in S$. Thus, $S \models C_1$.
2. $A = mi(e, [e|t], [f])$. Covered by $C_2 = mi(E, [E|_], [_])$. Note that $S \models C_2$.
3. $A = mi(e_i, [e_1, \ldots, e_i|t], [f_1, \ldots, f_{2i-1}])$, where $i > 1$. For (A) and (C) we should employ atom(s) from S, preferably with arguments which are subterms of those of A. A candidate argument is $u = [e_2, \ldots, e_i|t]$. Note that $mi(e_i, u, l) \in S$ whenever l is a list of length $2i - 3$. Hence, A is covered by $C_3 = mi(E, [_|U], [_, _|L]) \leftarrow mi(E, U, L)$. For (B) note that if $mi(e, u', l) \in S$, then each term $mi(e, [t_1|u'], [t_2, t_3|l])$ is in S. Hence, $S \models C_3$.

The obtained program is $\{C_1, C_2, C_3\}$. As it terminates for ground queries (we skip an easy proof), it is correct and complete w.r.t. (S, S).

2 Preliminaries

Basic notions. We assume that the reader is familiar with basics of logic programming and use the standard definitions and notation (Apt 1997; Apt and Bol 1994). A maybe nonstandard notion is (computed or correct) *answer*; by this we mean a query to which a (computed or correct) answer substitution has been applied. We assume a fixed set of function symbols (including constants) and of predicate symbols. $\mathcal{HU}$ stands for the Herbrand universe, and $\mathcal{HB}$ for the Herbrand base; $ground(P)$ is the set of ground instances of the clauses of a program P. $\mathbb{N}$ is the set of natural numbers. We deal with normal (Lloyd 1987) programs (called also "general" (Apt and Bol 1994)), and we usually call them just "programs." A procedure is a set of clauses beginning with the same predicate symbol.

Given a set $S \in \mathcal{HB}$ of ground atoms, $\neg S$ will stand for $\{\neg A \mid A \in S\}$. By a *level mapping* we mean a function $|\ |\colon S \to W$, where $S \subseteq \mathcal{HB}$ and $(W, \prec)$ is a well-ordered set (i.e. one in which there does not exist an infinite decreasing sequence).

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

4-valued logic. This logic plays an auxiliary role in the paper, and at the first reading the references to it may be skipped. We employ Belnap's 4-valued logic. The truth values are the subsets of the set of two standard truth values **t** and **f**. The subsets $\emptyset, \{\mathbf{t}\}, \{\mathbf{f}\}, \{\mathbf{t}, \mathbf{f}\}$ will be denoted by $\mathbf{u}, \mathbf{t}, \mathbf{f}, \mathbf{tf}$, respectively. The rationale for the four values is that a query, apart from succeeding (**t**) and failing (**f**), may diverge (**u**); additionally a specification may permit it both to succeed and to fail (**tf**). A 4-valued Herbrand interpretation is a subset of $\mathcal{HB} \cup \neg\mathcal{HB}$. (We will often skip the word "Herbrand.") If such interpretation does not contain both A and $\neg A$ (for any $A \in \mathcal{HB}$) then it is said to be 3-valued. Consider an interpretation $I \subseteq \mathcal{HB} \cup \neg\mathcal{HB}$, and a ground atom $A \in \mathcal{HB}$. The truth value $v \subseteq \{\mathbf{t}, \mathbf{f}\}$ of A in I is determined as follows: $\mathbf{t} \in v$ iff $A \in I$, and $\mathbf{f} \in v$ iff $\neg A \in I$. We may briefly say that A is v in I, for example, p is **tf** and q is **u** in $\{p, \neg p\}$. We write $I, \vartheta \models_4 F$ (respectively $I \models_4 F$) to state that the truth value of a formula F in an interpretation I and a variable valuation ϑ (resp. all variable valuations) contains **t**. When I is 3-valued, $I \models_3$ stands for $I \models_4$. The logical operations $\wedge, \vee$ are defined, respectively, as the glb (lub) in the lattice $(\{\mathbf{t}, \mathbf{f}\}, \preceq_t)$, where $\mathbf{f} \preceq_t \mathbf{u} \preceq_t \mathbf{t}$, $\mathbf{f} \preceq_t \mathbf{tf} \preceq_t \mathbf{t}$, and **tf**, **u** are incomparable. The negation is defined by $\neg\mathbf{t} = \mathbf{f}$, $\neg\mathbf{f} = \mathbf{t}$, $\neg\mathbf{u} = \mathbf{u}$, $\neg\mathbf{tf} = \mathbf{tf}$. See the work by Stärk (1996) or Fitting (1991) for the treatment of $\leftrightarrow$, and further explanations.

Using standard logic. In this work we prefer to avoid dealing with technical details of 4-valued logic. Instead we encode it in the standard 2-valued logic. Such approach is maybe less elegant, but it seems convenient to work within a well-known familiar logic. We extend the underlying alphabet by adding new predicate symbols, a distinct new symbol p' for each predicate symbol p (except for $=$). For the encoding, we use the following notation (Drabent and Martelli 1991; Drabent 1996).

Let $\mathcal{F}$ be a query, the negation of a query, a clause, or a program. Then $\mathcal{F}'$ is $\mathcal{F}$ with p replaced by p' in every negative literal of $\mathcal{F}$ (for any predicate symbol p). Similarly, $\mathcal{F}''$ is $\mathcal{F}$ with p replaced by p' in every positive literal. If $I \in \mathcal{HB}$ is a (2-valued) Herbrand interpretation then I' is the interpretation obtained from I by replacing each predicate symbol p by p'. For example for a clause $C = a \leftarrow \neg b, c$, we have $C' = a \leftarrow \neg b', c$, and $C'' = a' \leftarrow \neg b, c'$. Let $I = \{a, c\}$, $J = \{c\}$. Then $I \cup J' \models C'$ and $I \cup J' \not\models C''$ (as $J' = \{c'\}$). Consider a 4-valued interpretation $I = X \cup \neg(\mathcal{HB} \setminus Y)$ (where $X, Y \in \mathcal{HB}$). For a query Q, we have $I \models_4 Q$ iff $X \cup Y' \models Q'$ and $I \models_4 \neg Q$ iff $X \cup Y' \models \neg Q''$.

Negation in Prolog, Kunen semantics. Here we present a brief discussion, for missing details see, for example, the work by Apt and Bol (1994) or Doets (1994).

A natural way of adding negation to Prolog was negation as failure (more precisely, negation as finite failure, NAFF). This means deriving $\neg Q$ if a query Q finitely fails, that is, has a finite SLD-tree without answers. (In Prolog, $\backslash{+}Q$ fails if Q succeeds, and succeeds when Q terminates without any answer.) The corresponding operational semantics for normal programs is SLDNF-resolution (cf. Section 4.2).

The appropriate declarative semantics for NAFF was proposed by Kunen (Kunen 1987; Apt and Bol 1994). (We will call it *Kunen semantics*, *KS* or *completion semantics*.) It is defined by means of logical consequences of the program completion $comp(P)$

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

(Clark 1978) in a 3-valued logic of Kleene;[1] a query Q (resp. its negation $\neg Q$) is a result of a program P when $comp(P) \models_3 Q$ (resp. $comp(P) \models_3 \neg Q$). So soundness (of some operational semantics) w.r.t. KS means that if Q with P succeeds with a computed answer $Q\theta$ then $comp(P) \models_3 Q\theta$, and if Q fails then $comp(P) \models_3 \neg Q$. Completeness means the implications in the other direction.

Kunen semantics properly describes NAFF and SLDNF-resolution: First, SLDNF-resolution is sound w.r.t. the Kunen semantics, and is complete for wide classes of programs and queries. Also, it can be shown that the only reasons for incompleteness are floundering (cf. Section 4.2) or non-fair selection rule (Drabent 1996). Moreover, natural ways of augmenting SLDNF-resolution by constructive negation (Stuckey 1991; Drabent 1995) are sound and complete w.r.t. this semantics.

The well-founded semantics. The suitable declarative semantics for normal programs with negation as (possibly infinite) failure (NAF) is the well-founded semantics (WFS). It is given by a specific 3-valued *well-founded model* $WF(P)$ of a given program P (Van Gelder *et al.* 1991; Apt and Bol 1994). WFS is not computable – it is in general impossible to check whether an infinite tree fails. An operational semantics for WFS is SLS-resolution (cf. Section 6.2). SLS-resolution is sound w.r.t. WFS, and complete for non-floundering queries (Ross 1992; Apt and Bol 1994). NAF and SLS-resolution are approximately implemented by Prolog with tabulation. The methods proposed in this paper depend on soundness of SLDNF- and SLS-resolution, but not on their completeness.

3 Specifications for the context of negation

Here we show how approximate specifications from Section 1 can be used for normal programs. This section is independent from the choice of semantics for negation. When we do not deal with negation, a specification describes which atoms may succeed, and which ones have to succeed. Now we also need to specify which atoms should/may fail. This would require four interpretations, which is cumbersome. However it is rather natural to demand that the (ground) atoms required to succeed are not allowed to fail, and those not allowed to succeed should fail. So a pair of interpretations as a specification will be treated from two points of view, as a specification for correctness, and for completeness (Drabent and Miłkowska 2005).

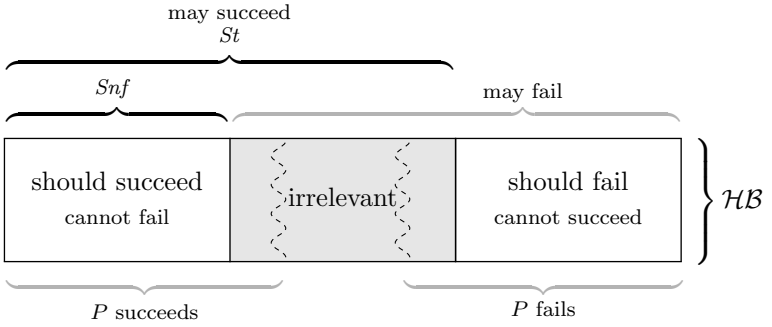
Definition 4.
An approximate **specification** is a pair (Snf, St) of Herbrand interpretations.
A specification (Snf, St) is called **proper** if $Snf \subseteq St$.

We usually drop the adjective "approximate." The letters nf, t stand for "not false," and "true" (cf. the 4-valued view of Definition 5). Comparing with Section 1, Snf corresponds to a specification for completeness, and St to that for correctness. The

1 The logic may be seen as Belnap's 4-valued logic without the truth value **tf**. Stärk (1996) proved that the logical consequences of $comp(P)$ are the same in the 3-valued and the 4-valued logic. For expressing Kunen semantics in the standard 2-valued logic see, for example, the paper by Drabent (1996) or by Stärk (1996) and references therein.

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

diagram illustrates a proper specification $spec = (Snf, St)$, and the ground atomic queries that succeed/fail for a program P which is correct and complete w.r.t. *spec*.



Considering program correctness, informal understanding of a specification (Snf, St) is that $A \in St$ means that A can be (an instance of) an answer of the program; and $A \in \mathcal{HB} \setminus Snf$ means that the query A can fail (and $\neg A$ can be an instance of an answer). Hence, any atom $A \in St \setminus Snf$ is allowed both to succeed and fail, and any $A \in Snf \setminus St$ is neither allowed to succeed, nor to fail.

So, in the notation introduced in Section 2, for any answer Q of a program P correct w.r.t. (Snf, St) we have $St \cup Snf' \models Q'$. Also, if Q with P fails then $St \cup Snf' \models \neg Q''$.

Considering program completeness, specification (Snf, St) requires any atom $A \in Snf$ to succeed, and any $A \in \mathcal{HB} \setminus St$ to fail. So $St \cup Snf' \models Q''$ means that Q is required to be program answer; $St \cup Snf' \models \neg Q'$ means that Q is required to fail. A non proper specification makes no sense in the context of completeness, as it requires some atoms to both succeed and fail.

As an example, consider specification (r, r_+) for a list membership predicate from Example 1. It is proper, and it says that the atoms from r (resp. r_+) must (may) succeed (hence those from $\mathcal{HB} \setminus r_+$ cannot succeed). Taken negation into account, it also states that the atoms from $\mathcal{HB} \setminus r$ (resp. $\mathcal{HB} \setminus r_+$) may (must) fail (hence those from r cannot fail).

The following definition provides a 4-valued logic view at approximate specifications.

Definition 5.
For a specification $spec = (Snf, St)$, let us define two 4-valued interpretations:
$I^4(Snf, St) = St \cup \neg(\mathcal{HB} \setminus Snf)$ (*spec* treated as a specification for correctness),
$I_4(Snf, St) = Snf \cup \neg(\mathcal{HB} \setminus St)$ (*spec* treated as a specification for completeness).

4 Correctness and completeness, Kunen semantics

In this section we present sufficient conditions for correctness and semi-completeness for normal programs under Kunen semantics.

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

4.1 Program correctness

Definition 6 (Correctness, Kunen semantics).
A program P is **correct** under KS w.r.t. a specification $spec = (Snf, St)$ if for any query Q

(i) if $comp(P) \models_3 Q$ then $St \cup Snf' \models Q'$ (equivalently, $I^4(Snf, St) \models_4 Q$),
(ii) if $comp(P) \models_3 \neg Q$ then $St \cup Snf' \models \neg Q''$ (equivalently, $I^4(Snf, St) \models_4 \neg Q$).

For an atomic query $Q = A$, conditions (i), (ii) reduce respectively to: if $comp(P) \models_3 A$ then $St \models A$, and if $comp(P) \models_3 \neg A$ then $Snf \models \neg A$.

Soundness of SLDNF-resolution relates this notion of correctness to actual computations: if Q is a computed answer for P then $comp(P) \models_3 Q$, and if Q finitely fails then $comp(P) \models_3 \neg Q$.

Definition 7 (covered atom, normal programs).
An atom $A \in \mathcal{HB}$ is **covered** by a clause C w.r.t. a specification $spec = (Snf, St)$ if there exists a ground instance $A \leftarrow \vec{B}$ of C such that $B \in Snf$ for each positive literal B from $\vec{B}$, and $B \notin St$ for each negative literal $\neg B$ from $\vec{B}$. (In other words, $St \cup Snf' \models \vec{B}''$).

A is covered w.r.t. spec by a program P if it is covered w.r.t. spec by a clause $C \in P$.

Speaking informally, A is covered by clause C if C is able to produce A out of literals required by *spec* to be produced by the program. As an example, we show that $m(a, [c, a])$ is covered by $C = m(E, [_|T]) \leftarrow m(E, T), \neg m(E, [b])$ w.r.t. specification (r, r_+) from Example 1. The body of the considered ground instance of C is L_1, L_2, where $L_1 = m(a, [a])$, $L_2 = \neg m(a, [b])$; then $(L_1, L_2)''$ is $m'(a, [a]), \neg m(a, [b])$ and $r_+ \cup r' \models (L_1, L_2)''$.

The following theorem (Drabent 2022) (it also follows from the paper by Ferrand (1993)) provides a way of proving correctness of normal programs under KS.

Theorem 8 (Correctness, Kunen semantics).
A program P is correct w.r.t. a specification $spec = (Snf, St)$ under KS if

1. $St \cup Snf' \models P'$, and
2. each atom $A \in Snf$ is covered by P w.r.t. *spec*.

Note that conditions 1 and 2 are, in a sense, a natural generalization of the sufficient conditions for correctness and semi-completeness for positive programs. The conditions actually imply correctness of P under Fitting (1985) semantics (which is implied by correctness under KS). See the paper by Drabent (2022) for a stronger sufficient condition. Examples of applying the theorem are contained in Examples 15, 16 below.

4.2 SLDNF-resolution

To generalize semi-completeness for normal programs under Kunen semantics, we need to refer to an operational semantics. To avoid including a rather lengthy definition of SLDNF-resolution, we make it explicit which aspects of it will be used here. In this way the proposed method is sound for any variant of SLDNF-resolution with the features specified below (e.g. those discussed by Apt and Doets (1994)).

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

Firstly, we assume that SLDNF-resolution is sound w.r.t. KS (cf. Section 2). We also assume that, for any program P, if Q finitely fails then any its instance $Q\theta$ finitely fails, and if Q succeeds with a most general answer (i.e. the answer $Q\eta$ is a variant of Q) then any its instance $Q\theta$ succeeds with a most general answer. Some negative literals are distinguished as *dealt with* (this may depend on the program and the selection rule). If $\neg A$ is dealt with, then A finitely fails or succeeds with a most general answer. Moreover, each ground $\neg A$ for which A finitely fails or succeeds is dealt with. Also, if $\neg A$ is dealt with, then any its instance $\neg A\theta$ is.

Secondly, we assume that – given a program P, a query Q and a selection rule $\mathcal{R}$ – the search space for Q contains the **main SLDNF-tree** for Q (shortly, main tree) as follows. (The search space may be a set of trees and successful derivations with assigned ranks (Lloyd 1987), the SLDNF-tree (Apt and Doets 1994), etc; for the approach of Lloyd (1987), the main tree is the SLDNF-tree for Q.)

1. The nodes of a main tree T for P and Q via $\mathcal{R}$ are queries. The root is Q, and $\mathcal{R}$ selects a literal in each non-empty node. The children of a node with a positive literal selected are as in an SLD-tree. Such node is marked *failed* if it has no children. For a node $\vec{A}, \neg A, \vec{B}$ with a negative literal $\neg A$ selected, one of (a), (b), (c) holds.

   (a) $\neg A$ is not dealt with, $\vec{A}, \neg A, \vec{B}$ is a leaf of T and is marked *floundered*,
   (b) $\neg A$ is dealt with, A succeeds with a most general answer, $\vec{A}, \neg A, \vec{B}$ is a leaf of T and is marked *failed*,
   (c) $\neg A$ is dealt with, A finitely fails, and $\vec{A}, \neg A, \vec{B}$ has a single child $\vec{A}, \vec{B}$.

   A branch of T is called an **SLDNF-derivation** for Q. Such derivation is successful, if its last query is empty.
2. T (and Q) **finitely fails** if T is finite and all its leaves are marked *failed*.
3. T **succeeds** (and Q succeeds) if T contains an empty leaf. Let θ be the composition of the mgu's (most general unifiers) along a successful branch D of T. (Note that there are no mgu's related to nodes with a negative literal selected.) Then $Q\theta$ is the computed **answer** of D for Q.

We say that a main tree T flounders if it contains a leaf marked *floundered*. A main tree is **not diverging** if it is finite, and does not flounder.

Obviously, in practice, it is not enough that the main tree is not diverging, we need that the whole computation terminates, that is the whole search space is finite. For a sufficient condition for termination, see the overview by Apt and Bol (1994) and the approach of Drabent and Miłkowska (2005, Section 4.3.5).

4.3 Completeness of normal programs under the Kunen semantics

We show that if a program is correct then it is semi-complete. The main theorem and the related lemmas follow some ideas of Drabent and Miłkowska (2005). First we generalize the definition of semi-completeness for normal programs with the Kunen semantics.

Definition 9 (Completeness, Kunen semantics).
A program P is **complete for a query** Q under KS w.r.t. a specification (Snf, St) if

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

(i) $St \cup Snf' \models Q''$ implies $comp(P) \models_3 Q$,
(ii) $St \cup Snf' \models \neg Q'$ implies $comp(P) \models_3 \neg Q$.

Program P is **complete** under KS w.r.t. *spec* if it is complete under KS w.r.t. *spec* for any query Q.

Note that the antecedents of (i), (ii) are respectively equivalent to $I_4(Snf, St) \models_4 Q$, and $I_4(Snf, St) \models_4 \neg Q$.

Definition 10.
A program P is **semi-complete** w.r.t. a specification *spec* under KS if P is complete under KS w.r.t. *spec* for any query Q for which there exists a non diverging main SLDNF-tree.

A program P is **SLDNF-semi-complete for a query** Q w.r.t. a $spec = (Snf, St)$ if for any non diverging main SLDNF-tree T for P and Q, and any instance $Q\theta$ of Q

(i) $St \cup Snf' \models Q''\theta$ implies that T contains an answer $Q\rho$ more general than $Q\theta$,
(ii) $St \cup Snf' \models \neg Q'$ implies that T finitely fails.

Program P is **SLDNF-semi-complete** w.r.t. *spec* if P SLDNF-semi-complete for each query Q.

In less formal wording, P is SLDNF-semi-complete if each non diverging main SLDNF-tree produces the results required by the specification. Note that SLDNF-semi-complete implies semi-complete (by soundness of SLDNF-resolution).

Example 11 (completeness for positive programs versus completeness under KS).
Consider a graph with nodes a, b, c and the set of edges given by a set of facts $G = \{ e(a,b), e(b,a) \}$. Consider a specification $spec = (St, St)$ where $St = G \cup St_p$ and $St_p = \{ p(a,b), p(b,a), p(a,a), p(b,b) \}$ (so $p/2$ describes which nodes in G are connected). Consider a program $P = G \cup \{ p(X,Y) \leftarrow e(X,Y).\ p(X,Z) \leftarrow e(X,Y), p(Y,Z). \}$.

Treated as a positive program, P is correct and complete w.r.t. St, as St is its least Herbrand model. Under the Kunen semantics, P is correct and SLDNF-semi-complete w.r.t. *spec*, but not complete. For instance, *spec* requires $p(a,c)$ and $p(c,a)$ to fail. Query $p(c,a)$ finitely fails (under Prolog selection rule), and $comp(P) \models_3 \neg p(c,a)$. On the other hand, $p(a,c)$ diverges (for any selection rule the SLD-tree is infinite), and $comp(P) \not\models_3 \neg p(a,c)$.[2]

When considering program completeness, we add a requirement on the underlying language. Speaking informally, a sufficient supply of symbols is necessary. By an **extended language** we mean the first order language over an alphabet containing an infinite set of constants, or a function symbol not occurring in the considered program and queries. In a sense, specifications in an extended language do not require too much. For instance,

2 To show that $comp(P) \not\models_3 \neg p(a,c)$, note that $p(a,c)$ is true in a 3-valued model of $comp(P)$, namely in $I = St_1 \cup \neg(\mathcal{HB} \setminus St_1)$ where $St_1 = St_p \cup \{p(a,c), p(b,c)\}$. To show that for any selection rule $p(a,c)$ diverges, note that it does not succeed (as otherwise P is incorrect w.r.t. *spec*), and it does not finitely fail (as otherwise $comp(P) \models_3 \neg p(a,c)$, contradiction).

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

when the only function symbol is $a/0$, and a specification is $spec = (\{p(a)\}, \{p(a)\})$, program $P = \{p(a).\}$ is not complete w.r.t. *spec*. This is because $p(X)$ is not an answer of P, but $\{p(a)\} \cup \{p'(a)\} \models p(X)$. This does not hold over an extended language.

For our main theorem we use two following lemmas. Their proofs are given in the supplementary material.

Lemma 12.
Let t be a term with $k \geq 0$ variables. Let f be a non-constant function symbol not occurring in t. Alternatively, let $c_1, \ldots, c_k$ be distinct constants not occurring in t. Then there exists an instance of t which is not unifiable with any term s such that (i) t is not an instance of s, and (ii) f (respectively any of $c_1, \ldots, c_k$) does not occur in s.

Lemma 13.
Let a normal program P be correct w.r.t. a specification $spec = (Snf, St)$ under KS. Let Q be a query. Assume an extended underlying language. Let T be a main SLDNF-tree for a query Q and P. If T is non diverging then T produces the results required by *spec* (i.e. T finitely fails if $St \cup Snf' \models \neg Q'$, and otherwise T has a computed answer more general than $Q\theta$ for each $Q\theta$ such that $St \cup Snf' \models Q''\theta$).

Theorem 14 (Semi-completeness, Kunen semantics).
Let a normal program P be correct w.r.t. a specification $spec = (Snf, St)$ under KS, over an extended language. Then P is SLDNF-semi-complete (hence semi-complete under KS) w.r.t. *spec*.

The theorem follows immediately from Lemma 13. As shown in supplementary material, Lemma 13 holds for ground queries and arbitrary languages. Hence, for any underlying language, correctness (of P w.r.t. *spec*) implies SLDNF-semi-completeness for ground queries (of P w.r.t. *spec*).

5 Systematic construction of programs, Kunen semantics

Theorems 8 and 14 suggest the following method: To build a program P which is correct and SLDNF-semi-complete (hence semi-complete) under the Kunen semantics w.r.t. a given specification $spec = (Snf, St)$, construct clauses so that

(A) each atom from Snf is covered by some clause $C \in P$ w.r.t. *spec*, and
(B) $St \cup Snf' \models C'$, for each constructed clause $C \in P$.

The program obtained in this way is correct w.r.t. *spec* (by Theorem 8) and SLDNF-semi-complete w.r.t. *spec* (by Theorem 14). For the program to be useful, care should be taken to (C) choose the clauses so that the program terminates and does not flounder for queries of interest. If the constructed program does not diverge for each ground query (under some selection rule) then it is complete w.r.t. *spec*.

Example 15.
We construct a program describing odd natural numbers. First let us provide a specification. We will use the usual representation, and say that a term $s^i(0)$ $(i \geq 0)$ is a number. So we need that all atoms from $Snf_o = \{\, o(t) \mid t \text{ is an odd number} \,\}$ are answers of the

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

program. We do not bother about terms not representing numbers, so our specification is $spec_o = (Snf_o, St_o)$ where $St_o = \{\, o(t) \in \mathcal{HB} \mid \text{if } t \text{ is a number then } t \text{ is odd} \,\}$. Note that $s(t) \in \mathcal{HB} \setminus St_o$ iff t is an even number; $spec_o$ requires each such $s(t)$ to fail, as expected.

We need to construct clauses so that each element of Snf_o is covered, and for each clause C we have $St_o \cup Snf'_o \models C'$. Let us use the fact that a number is odd iff it is a successor of a not odd one. This leads to $C = o(s(X)) \leftarrow \neg o(X)$. Note that all elements of Snf_o are covered by C w.r.t. $spec_o$ (as each odd number is of the form $s(t)$, the corresponding instance of C is $o(s(t)) \leftarrow \neg o(t)$, t is an even number, hence $o(t) \notin St_o$). Also, $St_o \cup Snf'_o \models C'$ (as if $Snf'_o \models \neg o'(t)$, then t is not an odd number, hence $s(t)$ is an odd number or not a number, thus $St_o \models o(s(t))$).

So $ODD = \{C\}$ is the constructed program, correct and semi-complete w.r.t. $spec_o$. ODD is complete, as it does not diverge for any ground query (each ground $o(s^i(u))$, where the main symbol of u is not s, succeeds or finitely fails, by induction on i).

Example 16 (Paths in a graph).
Consider a directed graph G with the edges described by a predicate $e/2$ (possibly given by a set of facts), correct and complete w.r.t.

$$(St_e, St_e) \qquad \text{where} \qquad St_e = \{\, e(t,u) \in \mathcal{HB} \mid \text{there is an edge } (t,u) \text{ in } G \,\}.$$

Let us construct a program finding paths between a given pair of nodes. We know that a naive program for transitive closure may miss some expected results for graphs with loops, due to negation by finite failure, cf. Example 11. To obtain a usable program, we add an argument with (a list of) nodes that should not be included in the path.

To construct a specification, let us introduce two auxiliary notions. By a simple path we mean one which does not visit a node twice (i.e. a $[t_1, \ldots, t_n]$ where $t_i \neq t_j$ for any $i \neq j$). We say that a path $[t_1, \ldots, t_n]$ bypasses a node e when $e \notin \{t_2, \ldots, t_{n-1}\}$. It bypasses a list of nodes if it bypasses every node from the list. Now consider

$$Snf_p = \left\{ p(t, v, s, [u_1, \ldots, u_n]) \in \mathcal{HB} \;\middle|\; \begin{array}{l} s \text{ is a simple path in } G \text{ from } t \text{ to } v \\ \text{bypassing } [u_1, \ldots, u_n] \end{array} \right\},$$

$$St_p = Snf_p \cup \{\, p(t, v, s, u) \in \mathcal{HB} \mid u \text{ is not a list} \,\}.$$

Our current specification for the program is $spec_0 = (Snf_p \cup St_e,\ St_p \cup St_e)$. Informally speaking, it implies that answers of the form $p(t, v, s, [\,])$ provide (all) the simple paths from t to v in G.

Procedure $e/2$ defining the graph is given, it remains to construct procedure $p/4$. For each atom from Snf_p we need a clause covering it. Consider an atom $A_1 = p(t, t, s, u) \in Snf_p$. Note that $s = [t]$, as the path is simple. A_1 is covered by a fact $C_1 = p(X, X, [X], _)$ (which satisfies (B), as $C'_1 = C_1$ and $St_p \models C_1$).

Consider an atom $A_2 = p(t, v, s, u) \in Snf_p$, where $t \neq v$. Then $s = [t|s_1]$ for a simple path s_1 from a node t_1 ($t_1 \neq t$) to v, so that (t, t_1) is an edge in G. As s bypasses u, $t_1 \notin u$ and s_1 bypasses u. Also, s_1 bypasses t, as otherwise s is not simple.

A candidate for a ground clause covering A_2 could be $C_{20} = p(t, v, [t|s_1], u) \leftarrow e(t, t_1), p(t_1, v, s_1, [t|u])$. However, it is easy to see that it would violate (B) (and lead to an incorrect program). We need to assure that $t \neq t_1$ and $t_1 \notin u$, in other words $t_1 \notin [t|u]$. For this it is convenient to employ a list membership program M from Example 2,

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

with specification $spec_m = (r, r_+)$, where $r = \{\, m(e_i, [e_1, \ldots, e_n]) \in \mathcal{HB} \mid 1 \le i \le n \,\}$, and $r_+ = r \cup \{\, m(e, u) \in \mathcal{HB} \mid u \text{ is not a list} \,\}$. Our specification becomes

$$spec = (Snf, St), \quad \text{where} \quad Snf = Snf_p \cup St_e \cup r, \quad St = St_p \cup St_e \cup r_+.$$

Consider a clause

$$C_2 = p(T, T1, [T|S], U) \leftarrow e(T, T2), \neg m(T2, [T|U]), p(T2, T1, S, [T|U]).$$

and its ground instance

$$C_{21} = \underbrace{p(t, v, [t|s_1], u)}_{H} \leftarrow \underbrace{e(t, t_1)}_{B_1}, \neg \underbrace{m(t_1, [t|u])}_{B_2}, \underbrace{p(t_1, v, s_1, [t|u])}_{B_3}.$$

Condition (A) holds: Assume $H = A_2$ and t_1 as above. Then C_2 covers A_2 w.r.t. *spec*, as $B_1, B_3 \in Snf$ and $B_2 \notin St$, due to the facts described above. (E.g. $B_3 \in Snf$ as s_1 is a simple path from t_1 to v; and s_1 bypasses t and the nodes in u.)

To show that C_{21} satisfies condition (B) (i.e. $St \cup Snf' \models C'_{21}$ for each ground instance C_{21} of C_2), assume that $B_1, B_3 \in St$ and $B_2 \notin Snf$. We have to show that $H \in St_p$. This is immediate if u is not a list. Otherwise $[t|u]$ is a list. Hence from $B_3 \in St$, we obtain that s_1 is a simple path from t_1 to v bypassing $[t|u]$. So $[t|s_1]$ is a path from t to v (by $B_1 \in St$). From $B_2 \notin Snf$ we obtain (as $[t|u]$ is a list) that $t_1 \notin [t|u]$. So $t_1 \neq t$ and thus path $[t|s_1]$ is simple. Also $[t|s_1]$ bypasses u, as the first element t_1 of s_1 does not occur in u and s_1 bypasses u. Hence, $H \in St_p$.

Thus the constructed program $PATH = \{C_1, C_2\} \cup M$ is correct and SLDNF-semi-complete w.r.t. *spec*,

We explain that, under the Prolog selection rule, the main SLDNF-tree for PATH and a query $Q_0 = p(t_0, t_1, s, u_0)$ (where t_0, u_0 are ground) does not diverge. Whenever $\neg m(\ldots)$ is selected, its arguments are ground, so the subsidiary tree (for $m(\ldots)$) is finite. This implies lack of floundering. On each path in the main tree, every third element is of the form $Q_{3i} = p(t_i, t_1, s_i, u_i)$ $(i \ge 0)$, where $t_i \notin u_i$ for $i > 0$, and $u_i = [t_{i-1}, \ldots, t_0|u_0]$. So $t_{i-1}, \ldots, t_0$ are distinct nodes of G, which is finite. Thus every such path is finite, the tree does not diverge, and thus *PATH* is complete w.r.t. *spec*.

6 The well-founded semantics

This section presents sufficient conditions for program correctness and semi-completeness under the well-founded semantics.

6.1 Program correctness, the well-founded semantics

A definition of correctness under WFS can be obtained from that of Definition 6 by replacing $comp(P) \models_3$ by $WF(P) \models_3$. Here is an equivalent and simpler formulation.

Definition 17.
A program P is **correct** under the well-founded semantics w.r.t. a specification $spec = (Snf, St)$ when $WF(P) \subseteq I^4(spec)$. In other words, $WF(P) \subseteq St \cup \neg(\mathcal{HB} \setminus Snf)$.

A program P is **complete** under the well-founded semantics w.r.t. a specification $spec = (Snf, St)$ when $I_4(spec) \subseteq WF(P)$. In other words $Snf \cup \neg(\mathcal{HB} \setminus St) \subseteq WF(P)$.

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

Theorem 18 (Correctness, WFS).
A program P is correct w.r.t. a specification $spec = (Snf, St)$ under the well-founded semantics provided that

1. $St \cup Snf' \models P'$, and
2. there exists a level mapping $|\ |: Snf \to W$ such that
   for each $A \in Snf$ there exists $A \leftarrow \vec{L} \in ground(P)$ such that
   (a) A is covered by $A \leftarrow \vec{L}$ (i.e. $St \cup Snf' \models \vec{L}''$),
   (b) for each positive literal L from $\vec{L}$, $|L| \prec |A|$.

The theorem is due to Ferrand and Deransart (1993) (they use different terminology, among others program correctness is divided into "partial correctness" and "weak completeness"). An informal explanation for condition 2b is that for each $A \in Snf$ it implies existence of a kind of a proof tree of finite height with root A.

Answer set programming (ASP) is outside the scope of this paper. We however mention the following property (see the supplementary material for a proof and example).

Proposition 19.
If condition 1 of Theorems 8, 18 holds for a program P then for each stable model I of P if $Snf \subseteq I$ then $I \subseteq St$.

6.2 SLS-resolution

SLS-resolution (Przymusinski 1989; Ross 1992; Apt and Bol 1994) is an operational semantics for WFS. Instead of its definition, we present its aspects of interest.

We assume that SLS-resolution is sound. This means that, for a program P, if a query Q fails then $WF(P) \models_3 \neg Q$, and if Q succeeds with answer $Q\theta$ then $WF(P) \models_3 Q\theta$. As previously, we assume that for any program P, if Q fails or succeeds with a most general answer, then the same holds for any its instance $Q\theta$.

We also assume that – given a program P, a query Q, and a selection rule $\mathcal{R}$ – the search space for Q contains the **main SLS-tree** (shortly, main tree), which is similar to a main SLDNF-tree of Section 4.2. The only difference is that a failed tree is not required to be finite. So the requirements are as previously, with "is finite" dropped from condition 2, "finitely fails" replaced by "fails," and "SLDNF-" by "SLS-". Also, T is not diverging if T does not flounder.

For example, the main SLS-tree for program $P_1 = \{\, p(s(X)) \leftarrow p(X). \,\}$ and query $p(X)$ fails; it consists of a single infinite branch. For $P_2 = \{\, a \leftarrow \neg a. \,\}$ and query a, the main SLS-tree T consists of two nodes a, $\neg a$, and the leaf $\neg a$ is marked *floundered*. (The "global SLS-tree"(Ross 1992; Apt and Bol 1994, Definition 9.9–9.11) has T as its root.)

6.3 Program completeness under the well-founded semantics

We begin, as previously, with introducing some technical notions.

Definition 20.
A program P is **complete for a query** Q under WFS w.r.t. a specification (Snf, St) if

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

(i) $St \cup Snf' \models Q''$ implies $WF(P) \models_3 Q$,
(ii) $St \cup Snf' \models \neg Q'$ implies $WF(P) \models_3 \neg Q$.

Note that a program is complete for any ground query iff it is complete (under WFS, w.r.t. a given specification).

Definition 21.
A program P is **semi-complete** w.r.t. a specification *spec* under WFS if P is complete under WFS w.r.t. *spec* for any query Q for which there exists a non diverging main SLS-tree. A program P is **SLS-semi-complete for a query** Q w.r.t. a $spec = (Snf, St)$ if for any non diverging main SLS-tree T for P and Q, and any instance $Q\theta$ of Q

(i) $St \cup Snf' \models Q''\theta$ implies that T contains an answer $Q\rho$ more general than $Q\theta$,
(ii) $St \cup Snf' \models \neg Q'$ implies that T fails.

Program P is **SLS-semi-complete** w.r.t. *spec* if P is SLS-semi-complete w.r.t. *spec* for each query Q.

By soundness of SLS-resolution, SLS-semi-completeness implies semi-completeness (of P w.r.t. *spec* under WFS).

As explained in the supplementary material, Lemma 13 holds also for WFS (with KS replaced by WFS, and SLDNF by SLS). From the lemma it immediately follows:

Theorem 22 (Semi-completeness, the well-founded semantics).
Let a normal program P be correct w.r.t. a specification $spec = (Snf, St)$ under WFS, over an extended language. Then P is SLS-semi-complete (hence semi-complete under WFS) w.r.t. *spec*.

As in the case of Theorem 14, if we drop the requirement on the language, then SLS-semi-completeness for ground queries is implied.

7 Construction of programs, the well-founded semantics

Theorems 18, 22 suggest the following method: To build a program P correct and SLS-semi-complete (hence semi-complete) under WFS w.r.t. a specification $spec = (Snf, St)$, choose a level mapping $|\ |: Snf \to W$ to a well-ordered set $(W, \prec)$, and construct clauses so that

(A) each atom $A \in Snf$
  (1) is covered by a ground instance $A \leftarrow \vec{L}$ of a constructed clause C, so that
  (2) for each positive literal L from $\vec{L}$, $|L| \prec |A|$,
(B) $St \cup Snf' \models C'$, for each constructed clause C.

For the obtained program to be useful, care should be taken to (C) choose the clauses so that the program does not diverge for the queries of interest. If the constructed program does not diverge for each ground query (under some selection rule), then it is complete w.r.t. *spec*.

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

Example 23.
We construct a procedure $p/2$ finding, under the well-founded semantics, whether two nodes of a given finite directed graph G are connected. The specification is

$$\begin{aligned} spec = (Snf, St), \text{ where } St = Snf = St_e \cup St_p, \\ St_e = \{\, e(t,u) \in \mathcal{HB} \mid \text{there is an edge } (t,u) \text{ in } G \,\}, \\ St_p = \{\, p(t,u) \in \mathcal{HB} \mid \text{there is a path from } t \text{ to } u \text{ in } G \,\}. \end{aligned}$$

Treating St_e as a set of facts provides a procedure $e/2$ correct and complete w.r.t. *spec*. To construct $p/2$ let us first choose the level $|p(t,u)|$ to be the length of the shortest path from t to u (for any $p(t,u) \in St_p$), and let $|e(t,u)| = 0$ (for any $e(t,u) \in St_e$). Now we need to construct clauses so that each atom $p(t,u) \in St_p$ is covered. A border case is $t = u$, for this we use a fact $C_1 = p(X,X)$ (for which (A)(1), (A)(2), (B) obviously hold). For $t \neq u$, there exists an edge (t,s) to a node s connected with u. This suggests a clause

$$C_2 \;=\; p(X,Z) \leftarrow e(X,Y), p(Y,Z).$$

Note that (B) holds for C_2. Consider a path of length $|p(t,u)|$ from t to u, and its first edge (t,s). Obviously, there exists in G a path from s to u of length $|p(t,u)| - 1$; so $p(s,u) \in St_p \subseteq Snf$. Also, $e(t,s) \in St_e \subseteq Snf$. Thus, $p(t,u)$ is covered by an instance $C_{20} = p(t,u) \leftarrow e(t,s), p(s,u)$ of C_2; (A)(2) holds for C_{20} as $|p(s,u)| < |p(t,u)|$.

The constructed program $PATH2 = St_e \cup \{C_1, C_2\}$ is correct and SLS-semi-complete w.r.t. *spec*. It is also complete w.r.t. *spec*, as diverging main SLS-trees for atomic queries do not exists, due to lack of negation in the program. In particular, $p(t,u)$ would fail for any not connected nodes t, u, even if the graph contains loops. Note that for such graphs PATH2 is not complete w.r.t. *spec* under the Kunen semantics.

8 Conclusions

This paper introduces two methods of constructing normal programs which are correct and semi-complete w.r.t. given specifications. The methods deal, respectively, with the 3-valued completion semantics of Kunen, and the well-founded semantics. (The former corresponds to Prolog, the latter to Prolog with tabulation.) The approach is declarative, one reasons in terms of programs seen as logical formulae, and abstracts from operational semantics. The methods are simple, each is described in some 1/4 page (cf. the beginnings of Sections 5, 7). They however do not formalize how to assure non-divergence of the constructed program. The methods are directly derived from sufficient conditions for correctness and semi-completeness. The approach uses, in a sense, a 4-valued logic; however everything is encoded in the standard logic. In particular, a specification is a pair of 4-valued Herbrand interpretations represented as two 2-valued ones. An important feature is that the specifications are approximate, they do not exactly describe the program semantics.

This work does not deal with ASP; roughly speaking because our specifications describe single literals, while in ASP the outcome of programs are sets of literals (stable models). However, sufficient conditions used here lead to a maybe useful characterization of stable models (Proposition 19).

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

The simplicity of the methods is partly due to introducing semi-completeness (i.e. completeness for non diverging queries). This divides completeness proofs into those for semi-completeness and non-divergence. Then we get semi-completeness for free, as it is implied by correctness. This (possibly surprising) property holds for both semantics; note however the dual understanding of specifications, see Section 3. Another interesting fact is that the method for Kunen semantics is basically the same as that for the case without negation (Drabent 2018).

In the author's opinion, the proposed methods can be used, possibly at some informal level, in practical everyday programming. They should be useful in teaching logic programming. The employed sufficient conditions for correctness and semi-completeness are of separate interest. They show how to reason declaratively about program properties. Again, they can be used in practical programming, and in teaching logic programming. For logic programming to be considered a useful declarative programming paradigm, it is necessary to have declarative methods for reasoning about programs, and for constructing programs. This work contributes to providing such methods.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068425100239.

Competing interests

The author declares none.

References

Apt, K. and Bol, R. 1994. Logic programming and negation: A survey. *The Journal of Logic Programming* 19-20, 9–71.

Apt, K. and Doets, K. 1994. A new definition of SLDNF-resolution. *The Journal of Logic Programming* 18, 177–190.

Apt, K. R. 1997. *From Logic Programming to Prolog*. Prentice-Hall.

Clark, K. L. 1978. Negation as failure. In *Logic and Databases*, H. Gallaire and J. Minker, Eds. Plenum.

Clark, K. L. (1979). Predicate logic as computational formalism. Tech. Rep. 79/59 TOC, Department of Computing, Imperial College, London. December.

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

Doets, K. 1994. *From Logic to Logic Programming*. MIT Press.

Drabent, W. 1995. What is failure? An approach to constructive negation. *Acta Informatica* 32, 1, 27–59.

Drabent, W. 1996. Completeness of SLDNF-resolution for non-floundering queries. *The Journal of Logic Programming* 27, 2, 89–106.

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

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press

Drabent, W. 2018. Logic + control: On program construction and verification. *Theory and Practice of Logic Programming* 18, 1, 1–29.

Drabent, W. 2022. On correctness of normal logic programs. In Logic-Based Program Synthesis and Transformation (LOPSTR 2022), Proceedings, A. Villanueva, Ed. Lecture Notes in Computer Science, Vol. 13474. Springer, 142–154,

Drabent, W. and Martelli, M. 1991. Strict completion of logic programs. *New Generation Computing* 9, 69–79.

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

Ferrand, G. 1993. The notions of symptom and error in declarative diagnosis of logic programs. In AADEBUG'93, Proceedings, P. Fritszon, Ed. LNCS, Vol. 749. Springer, 40–57.

Ferrand, G. and Deransart, P. 1993. Proof method of partial correctness and weak completeness for normal logic programs. *The Journal of Logic Programming* 17, 265–278.

Fitting, M. 1985. A Kripke-Kleene semantics for logic programs. *The Journal of Logic Programming* 2, 4, 295–312.

Fitting, M. 1991. Bilattices and the semantics of logic programming. *The Journal of Logic Programming* 11, 2, 91–116.

Kunen, K. 1987. Negation in logic programming. *The Journal of Logic Programming* 4, 4, 289–308.

Lloyd, J. W. 1987. *Foundations of Logic Programming*, Second, extended ed. Springer.

Malfon, B. 1994. Characterization of some semantics for logic programs with negation and application to program validation. In Proceedings of the 1994 International Logic Programming Symposium, M. Bruynooghe, Ed. MIT Press, 91–105.

Przymusinski, T. C. 1989. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning* 5, 167–205.

Ross, K. A. 1992. A procedural semantics for well founded negation in logic programs. *The Journal of Logic Programming* 13, 1–22.

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

Stärk, R. F. 1996. From logic programs to inductive definitions. In *Logic: From Foundations to Applications, European Logic Colloquium '93*, W. A. Hodges et al., Ed. Clarendon Press, Oxford, 453–481.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog*, 2 ed. The MIT Press. Available at https://mitpress.mit.edu/books/art-prolog-second-edition.

Stuckey, P. 1991. Constructive negation for constraint logic programming. In *6th Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, IEEE Computer Society, 328–339.

Van Gelder, A., Ross, K. A. and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38, 3, 620–650.

https://doi.org/10.1017/S1471068425100239 Published online by Cambridge University Press