Verification of logic programs

Dino Pedreschi *, Salvatore Ruggieri

Dipart. di Informatica, Universita di Pisa, Corso Italia 40, 56100 Pisa, Italy

Received 10 February 1997; received in revised form 31 August 1998; accepted 11 September 1998

Abstract

We present a proof method in the style of Hoare's logic, aimed at providing a unifying framework for the verification of total correctness of logic and Prolog programs. The method, which relies on purely declarative reasoning, has been designed as a trade-off between expressiveness and ease of use. On the basis of a few simple principles, we reason uniformly on several properties of logic and Prolog programs, including partial correctness, call patterns, absence of run-time errors, safe omission of the occur-check, computed instances, termination and modularity of program development. We finally generalize the method to general programs.

Keywords: Verification; Proof theory; Logic programs; Prolog; Correctness; Total correctness; Partial correctness

1. Introduction

The family of logic programming languages is advocated as an ideal support to declarative programming – an endeavor where programmers write specifications that can be directly used as programs. This ideal situation, however, is usually contradicted by practical experience. On the one hand, direct execution of specifications may be hopelessly inefficient, and, on the other hand, logic programming systems often exhibit slightly different semantics. For these reasons, declarative programs may fail to terminate, may end in run-time errors, may deliver unexpected output, may behave differently in different implementations.

It is therefore important to assess the correctness of a logic program with respect to its specification, or intended interpretation: a problem that has received particular attention in the recent years, as witnessed by the body of research cited in the Related Work section. Many proof methods and techniques have been put forward to address the various verification issues, including:

(i) partial correctness.

*Corresponding author. Tel.: +39 050 887252; fax: +39 050 887226; e-mail: pedre@dii.unipi.it
(ii) characterization of call patterns,
(iii) characterization of correct and computed instances,
(iv) universal termination,
(v) absence of type and run-time errors.
(vi) safe omission of the occur-check.
(vii) modular program development.

However, no comprehensive framework has been proposed, capable of addressing the various verification issues within a single proof theory on the basis of a few simple, unifying principles. For instance, Apt's book [8] presents a number of different techniques, each devoted to the analysis of specific issues by means of specific tools. A striking comparison naturally arises with imperative programming, where Hoare's logic thoroughly encompasses verification of sequential programs and provides the basis for verifying concurrent and distributed programs (see, e.g., Apt and Olderog [12].)

This paper introduces a proof theory designed as a candidate unifying framework for the verification of logic programs. The starting point of the research reported in this paper has been the recognition of a few core principles, common to several existing proof methods for logic programs. On this basis, a thorough proof theory has been developed, capable of addressing a reasonably large spectrum of properties for a reasonably large class of programs. The spectrum of properties is (i–vii) above, whereas the class of programs is that of logic programs, possibly with negation and arithmetic built-in's, which are designed to be executed according to a fixed selection rule. As a consequence, the proposed proof theory is general enough to encompass verification of Prolog programs.

The proof theory is based on Hoare's style triples \( \langle \text{Pre} \rangle \ P \langle \text{Post} \rangle \) which, for a logic program \( P \), specify the admissible input and expected output by means of pre- and postconditions. Although the logic programming version of a triple is defined on purely logical terms, it can be readily applied to reason about operational and run-time properties, thus abstracting away from the subtleties of the procedural interpretation of logic programming—unification and the logical variable, the search strategy, to mention a few. In this sense, the proposed verification method is carefully designed as a compromise between generality and expressiveness from the one side, and ease of use from the other side.

Technically speaking, the proof theory is obtained as a combination of (modifications of) existing proposals: the proof method for partial correctness of Bossi and Cocco [16], and the proof method for termination of Apt and Pedreschi [13]. The advantage of this operation is that the expressiveness of the combined method strictly exceeds the expressiveness of the separated methods both from a theoretical and a practical perspective. From a theoretical perspective, the classes of programs and properties addressed by the combined method is strictly larger than those of the separated methods. This claim is substantiated in the related work section. From a practical perspective, the combined method supports shorter and simpler proofs, with respect to the separated methods. In fact, fewer proof obligations are required in the combined method, and reasoning is performed in purely logical terms—i.e., in the logic programming jargon, at the level of ground objects—abstracting away from the complications of the logical variable.

On the basis of the above discussion, the original contribution of this paper is the introduction of a proof relation \( \vdash \langle \text{Pre} \rangle \ P \langle \text{Post} \rangle \) for total correctness, capable of addressing properties (i–vii) for logic programs with negation and arithmetic built-
in's. For obvious reasons of presentation, the \( \vdash \) proof method is introduced in an incremental way, by a step-wise definition of increasingly higher levels of verification, from a weak form of partial correctness up to full-fledged total correctness. This is a standard presentation style adopted in many textbooks on Hoare’s logic for imperative programming, such as Ref. [12]. It is worth noting that certain fragments of the proposed method are not new, as discussed in Section 6. However, the complete method is new both in the spectrum of addressed properties and in the class of addressed programs.

1.1. Plan of the paper

In Section 2, we discuss which notions of semantics and specifications are appropriate for program verification, and adopt a variant of the least Herbrand model semantics. The proof method based on triples is introduced in Section 3. We show how to reason about modular proofs, correctness, call patterns, correct and computed instances, and termination. In Section 4, we discuss refinements of the proof method to deal with arithmetic built-in's and modular program development, and some results on the safe omission of the occur-check, verification of meta-interpreters and decidability issues. Finally, in Section 5, the method is extended to reason on general logic programs.

1.2. Notation

A word on terminology is in order. Throughout the paper we use the standard notation of logic programming, as in Refs. [7,41], unless specified otherwise. In particular, we use queries instead of goals. LD(NF)-resolution is SLD(NF)-resolution with the leftmost selection rule. A language \( L \) is a pair \( \langle \Sigma_L, \Pi_L \rangle \) of (not necessarily disjoint) set of function and predicate symbols. To each symbol a non-negative arity is assigned. Ambivalent syntax is allowed, in the sense that function and predicate symbols may overlap [40]. We consider a fixed language \( L \) in which programs and queries are written. In other words, all the results are parametric with respect to \( L \), provided \( L \) is rich enough to contain every symbol of the programs and queries under consideration.

\( \text{Atom}_L \) is the set of atoms on the language \( L \). \( U_L \) the Herbrand universe of \( L \), and \( B_L \) is the Herbrand base of \( L \). \( L_P \) is the language generated by the program \( P \). \( B_P \) is an abbreviation for \( B_{L_P} \). By \( M^L_P \) we denote the least Herbrand model of \( P \) with \( L \) as the underlying language. \( FF^L_P \) is the finite failure set of \( P \), and \( FF^L_P \) is \( B_L \setminus FF^L_P \). An atom is called pure if it is of the form \( p(x_1, \ldots, x_n) \) where \( x_1, \ldots, x_n \) are different variables. \( \text{rel}(A) \) denotes the predicate symbol of the atom \( A \).

We write \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \) iff \( A \leftarrow B_1, \ldots, B_n \) is a ground instance of a clause from \( P \). Given a Herbrand interpretation \( I \) and a query \( Q \) we write \( I \models Q \) if \( I \models Q \) is a model of \( Q \). In particular, if \( A \) is a ground atom then \( I \models A \) iff \( A \in I \). Finally, \( T_P \) is the classical immediate consequence operator defined as follows: \( T_P(I) = \{ A \in B_L \mid \exists A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) : I \models B_1, \ldots, B_n \} \).

We adopt the Prolog notation when writing lists. For \( z \subseteq U_L \), \( \text{List}(z) \) denotes the set of ground lists whose elements belong to \( z \). \( \mathcal{GList} \) is the set of ground lists, i.e. \( \text{List}(U_L) \). The list-length function from ground terms into natural numbers is defined as follows: \( |f(\ldots)| = 0 \) if \( f \) is not the list constructor symbol \( [\ldots] \), and \( |x[t]| = |t| + 1 \) otherwise.
The following assumptions on notation will be useful:

- an identifier with lower-case initial letter, such as \( x, y, z, xs, ys, zs \), is a meta-variable for ground terms;
- an identifier with upper-case initial letter, such as \( X, Y, Z, Xs, Ys, Zs \), is a meta-variable for (not necessarily ground) terms;
- symbols and expressions from the underlying language \( L \) are written in the type-writer style, such as \( \text{member}(x), X, Xs, Y \).

With these assumptions, \( x \) is a ground term, \( X \) a term, and \( X \) a logic variable.

2. Declarative programming

2.1. Which semantics for program verification?

Of course, when verifying a logic program \( P \), it would be helpful to use its declarative semantics. However, several declarative semantics have been proposed as promising alternatives to the least Herbrand model \( M^P_b \) (also known as \( \mathcal{H} \)-semantics) for supporting program verification in a natural way. Among the others, we mainly focus on two of them, namely the \( \mathcal{C} \)-semantics of Falaschi et al. [35] (also known as the least term model of Clark [22]) and the \( \mathcal{S} \)-semantics of Falaschi et al. [18,34].

Definition 2.1. For a logic program \( P \) we define:

\[
\mathcal{H}(P) = \{ A \in B_L \mid P \models A \}.
\]

\[
\mathcal{C}(P) = \{ A \in \text{Atom}_L \mid P \models A \}.
\]

\[
\mathcal{S}(P) = \{ A \in \text{Atom}_L \mid A \text{ is a computed instance of a pure atom} \}.
\]

In Apt et al. [9], the relative information ordering of the three semantics is studied and the semantics are related to each other with the claim that for a large class of programs and properties one can restrict to consider only the simple \( \mathcal{H} \)-semantics. Under certain conditions, the \( \mathcal{C} \)- and \( \mathcal{S} \)-semantics can be reconstructed starting from the least Herbrand model. Therefore, \( \mathcal{H} \)-semantics seems to be a good trade-off between expressiveness, abstraction and complexity of use in paper & pencil proof methods. Then, a natural approach consists of considering \( M^P_b \) as the intended specification — therefore, the verification of a program is viewed as checking that the intended specification of program and its least Herbrand model do coincide.

This approach, however, turns out to be inadequate: strangely enough, the least Herbrand models semantics is not sufficiently abstract. In fact, the absence of types implies that the least Herbrand model is generally polluted with unintended atoms.

Consider the APPEND program taken from Sterling and Shapiro [54]:

\[
\text{append}(Xs, Ys, Zs) \quad \text{— } Zs \text{ is the concatenation of lists } Xs \text{ and } Ys.
\]

\[
\text{append}([], Xs, Xs).
\]

\[
\text{append}([X|Xs]. Ys. [X|Zs]) \quad \text{—}
\]

\[
\text{append}(Xs, Ys, Zs).
\]

APPEND is intuitively correct with respect to its specification but (if there are sufficiently many symbols in the language) its intended interpretation is not a model of the program. In fact, in the least Herbrand model unintended atoms
appear, such as `append([ ], foo, foo)`. For efficiency reasons run-time type-checks are dropped.

As a consequence, reasoning about the whole least Herbrand model implies having to take into account ill-typed atoms, thus making the specification complex and counter-intuitive. This problem becomes much harder in modular program development, since adding more symbols to the language in the upper modules entails changing the least Herbrand model of lower modules, and hence their correctness properties. A clear point emerges from the previous discussion: a semantics for verification should take the intended or well-typed queries into account.

### 2.2. Specifications and semantics

Following a Hoare’s logic style of defining partial and total correctness, we stipulate that a specification is a pair $( Pre, Post )$ of Herbrand interpretations, i.e., subsets of $B_{L}$. The rationale under this choice is the following. The first interpretation, $Pre$, specifies the intended, or well-typed one-atom queries, i.e., those queries for which we designed the program under consideration. The second interpretation, $Post$, specifies some desired property of successful one-atom queries. In this sense, a specification $(Pre, Post)$ describes the input–output behavior of a logic program, in a way that closely resembles that in Hoare’s logic, where preconditions specify the admissible input, and postconditions specify (properties of) the expected output. Here, preconditions specify the admissible input queries, and postconditions specify the expected output, namely properties of the correct instances of the input queries.

According to this choice, the well-typed fragment of the least Herbrand model is $M_{p}^{p} \cap Pre$. We are now ready to define our notions of (weak) partial and (weak) total correctness.

#### Definition 2.2

Let $P$ be a logic program.

- $P$ is partially correct w.r.t. a specification $(Pre, Post)$ iff $M_{p}^{p} \cap Pre = Post$.
- $P$ is totally correct w.r.t. a specification $(Pre, Post)$ iff $M_{p}^{p} \cap Pre = Post$ and $Pre \subseteq M_{p}^{p} \cup FF_{p}^{p}$, where $FF_{p}^{p}$ is the finite failure set of $P$.

In addition, $P$ is weak partially or weak totally correct if the weaker requirement $M_{p}^{p} \cap Pre \subseteq Post$ holds instead of $M_{p}^{p} \cap Pre = Post$.

As a consequence of this definition, partial correctness of a program $P$ w.r.t. a specification $(Pre, Post)$ entails that $Post$ coincides with the well-typed fragment of $M_{p}^{p}$. As usual, the difference between partial and total correctness is that, in the latter case, we also require a weak form of termination, namely that every query in $Pre$ either succeeds or (finitely) fails. Although $Pre$ is a set of ground atoms, it should be stressed that admissible queries are not required to be necessarily ground. We shall devise proof methods to reason about any atomic query $Q$ such that $Pre \models Q$, namely any query true in $Pre$, or equivalently, any query whose instances are included in $Pre$.

It should be noted that both partial and total correctness are defined in purely declarative terms, as the sets $M_{p}^{p}$ and $FF_{p}^{p}$ can be constructed without reference to the procedural interpretation of logic programming [8,41]. In addition to the mentioned declarative notions, the mechanism of pre- and post-conditions is suitable...
to deal with the operational notion of call pattern characterization. In other words, we require that every atom selected during a derivation starting from an intended query is fully characterized by \( \text{Pre} \). In such a case, \( \text{Pre} \) can be used to specify certain desired run-time properties, ranging from persistency of types up to absence of run-time errors, safe omission of the occur-check and, for general programs, non-floundering.

As an example, the \textsc{Append} program is intuitively totally correct w.r.t. the specification:

\[
\begin{align*}
\text{Pre}_{\text{Append}} &= \{ \text{append}(\mathit{xs}, \mathit{ys}, \mathit{zs}) \mid \mathit{xs}, \mathit{ys} \in \text{GList} \}, \\
\text{Post}_{\text{Append}} &= \{ \text{append}(\mathit{xs}, \mathit{ys}, \mathit{zs}) \mid \mathit{xs}, \mathit{ys} \in \text{GList} \land \mathit{zs} = \mathit{xs} \ast \mathit{ys} \}.
\end{align*}
\]

where \( \ast \) is the list concatenation operator. Moreover, \( \text{Pre}_{\text{Append}} \) characterizes the call patterns of the queries where \text{append} is called with the first two arguments filled in with (not necessarily ground) lists.

Notice that the weak version of either notions of correctness entails that \( \text{Post} \) specifies some property of \( M^p \cap \text{Pre} \). For instance, the \textsc{Append} program is weak totally correct w.r.t. \( (\text{Pre}_{\text{Append}}, \text{Post}) \), where:

\[
\begin{align*}
\text{Post} &= \{ \text{append}(\mathit{xs}, \mathit{ys}, \mathit{zs}) \mid |\mathit{zs}| = |\mathit{xs}| + |\mathit{ys}| \},
\end{align*}
\]

and \(|.|\) is the list-length function. Therefore, a postcondition in the sense of the weak correctness describes a property of every correct instance of an intended query.

It is worth noting that the definitions of (weak) partial and (weak) total correctness are stated in full general terms. As a consequence of our commitment to the study of fixed selection rules, universal termination and call pattern characterization, the resulting proof method is a sufficient method for the notions of correctness mentioned above. Once again, our objective is to design a method that is a trade-off between expressiveness (i.e., the class of programs and properties it is able to reason about) and ease of use in paper & pencil proofs.

3. Proof theory

3.1. The proof method

We now introduce a proof method for the various notions of correctness, by means of the concept of (Hoare's logic style) triples \( \{\text{Pre}\} P \{\text{Post}\} \) (for programs \( P \)) and \( \{\text{Pre}\} Q \{\text{Post}\} \) (for queries \( Q \)). Triples are the basic tools to prove correctness. The proof method essentially consists in the definition of a proof relation \( \vdash \), for triples, which, as we will show later, provides a tool for reasoning about (weak) total correctness. As discussed in the introduction, we shall study \( \vdash \), in an incremental way, by considering first a sub-relation \( \vdash \) on triples, which provides a tool for reasoning about (weak) partial correctness.

The next key definition introduces both proof relations, \( \vdash_{\ell} \) and \( \vdash \), by explaining the proof obligations needed to prove a triple in either sense. First, we recall from Bezem [14] the notion of level mapping (on a language \( L \)).

**Definition 3.1.** A level mapping (on \( L \)) is a function \( |.| : B_L \rightarrow N \) of ground atoms to natural numbers.
Definition 3.2. Consider a program \( P \), a query \( Q \), and a specification \( (\text{Pre}, \text{Post}) \). We write:

- \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) iff there exists a level mapping \( I \) such that for every \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_I(P) \):
  1. for \( i \in [1..n] \):
     \[
     \text{Pre} \models A \land \text{Post} \models B_1, \ldots, B_{i-1} \Rightarrow
     \begin{align*}
     (a) & \quad \text{Pre} \models B_i \quad \text{and} \quad \\
     (b) & \quad |A| > |B_i|.
     \end{align*}
     \]
  2. \( \text{Pre} \models A \land \text{Post} \models B_1, \ldots, B_n \Rightarrow \text{Post} \models A \).

   We write \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) when (1a) and (2) hold. \( \text{Pre} \) is called a precondition and \( \text{Post} \) a postcondition.

- \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) iff for every ground instance \( A_1, \ldots, A_n \) of \( Q \):
  3. for \( i \in [1..n] \) \( \text{Post} \models A_1, \ldots, A_{i-1} \Rightarrow \text{Pre} \models A_i \).

- \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) iff there exist a level mapping \( I \) and \( k \in \mathbb{N} \) such that for every ground instance \( A_1, \ldots, A_n \) of \( Q \):
  4. for \( i \in [1..n] \) \( \text{Post} \models A_1, \ldots, A_{i-1} \Rightarrow \text{Pre} \models A_i \land k > |A_i| \).

Intuitively, for a clause \( C \) in \( \text{ground}_I(P) \) there are \( n + 1 \) proof obligations to conclude that the relation \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) holds:

1. each atom \( B \) in the body of \( C \) is in \( \text{Pre} \) when the the head of \( C \) is in \( \text{Pre} \) and all the atoms to the left of \( B \) in the body of \( C \) are in \( \text{Post} \);
2. the head of \( C \) is in \( \text{Post} \) when it is in \( \text{Pre} \) and all the atoms in the body of \( C \) are in \( \text{Post} \).

In the case of \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) the decreasing of the level mapping is also required, i.e. the level mapping plays the role of a termination function. Strictly speaking, the level mapping has to be defined only on the precondition \( \text{Pre} \).

The left-to-right propagation of assumptions in proof obligations is biased by the left-to-right evaluation strategy of Prolog, in the sense that we require that a body atom is ready to be executed (i.e., it is in \( \text{Pre} \)) when the atoms to its left have been already executed (i.e., they are in \( \text{Post} \)). However, the presented notion is purely declarative, and no procedural intuition is needed to carry on the proof obligations. Moreover, we observe that the proof method applies to arbitrary fixed selection rules other than leftmost's by simply considering permutations of the body atoms.

Proving that a triple is in the relation \( \vdash \) or \( \vdash \), for a given program or query involves reasoning on their ground instances only. Basically, the definition provides a standard way for lifting up the results to non-ground queries. The advantage is that this lifting is made a posteriori.

Finally, we point out that Definitions 3.2 (3,4) for a query \( Q \) are derived from Definitions 3.2 (1,2) by considering the program \( \text{ans} \leftarrow Q \), where \( \text{ans} \) is a fresh predicate. This follows from the following useful relation, which is immediate from Definition 3.2 (1,2): for \( \vdash \{\text{Pre}\} P \{\text{Post}\} \) and \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_I(P) \)

\[
A \in \text{Pre} \quad \text{implies} \quad \vdash \{\text{Pre}\} B_1, \ldots, B_n \{\text{Post}\}.
\]

As explained in the introduction, the ultimate goal of this paper is to show that the relation \( \vdash \), yields a proof theory for total correctness of (general) logic programs. The study of relation \( \vdash \), is performed by means of some intermediate steps. First, we study how the subrelation \( \vdash \) allows us to reason on (weak) partial correctness and to characterize call patterns; second, we study how \( \vdash \) allows us to characterize
correct and computed instances of intended queries; and finally, we study how the \( \vdash \) relation completes the picture by dealing with termination.

### 3.1.1. Example: preorder tree traversal

As an example to clarify the form of the needed proof obligations, consider the program \textsc{preorder}:

\[
\text{preorder}(T, Ls) \leftarrow \\
\text{Ls} \text{ is a preorder traversal of the binary tree } T \\
\text{(p1)} \text{ preorder}(\text{nil}, [ ]). \\
\text{(p2)} \text{ preorder}(\text{leaf}(X), [X]). \\
\text{(p3)} \text{ preorder}(\text{tree}(X, \text{Left}, \text{Right}), Ls) \leftarrow \\
\text{preorder}(\text{Left}, \text{As}), \\
\text{preorder}(\text{Right}, \text{Bs}), \\
\text{append}(\text{[X|As]}, \text{Bs}, \text{Ls}).
\]

augmented with the \textsc{append} program. The set of ground binary trees \( \text{Tree}(\alpha, \beta) \) whose leaves belong to \( \alpha \subseteq U_L \) and intermediate nodes belong to \( \beta \subseteq U_I \) is defined by the grammar:

\[
\text{Tree ::= nil | leaf(x) | tree(\beta, Tree, Tree)}
\]

For instance, if \( \alpha = \{0, 1, 2, \ldots \} \) and \( \beta = \{+, -, \times, \ldots \} \), we have that \( \text{Tree}(\alpha, \beta) \) is the set of syntax trees defining arithmetic expressions on natural numbers. We denote by \( ||t|| \) the number of nodes of a tree \( t \), defined as follows:

\[
||\text{leaf}(x)|| = 1 \\
||\text{tree}(x, l, r)|| = ||l|| + ||r|| + 1 \\
||f(t_1, \ldots, t_n)|| = 0 \text{ otherwise.}
\]

Intuitively, an intended use of \textsc{preorder} is to compute the preorder traversal of a given tree. This is formally expressed by defining:

\[
\text{Pre}_{\textsc{preorder}} = \{\text{preorder}(t, Ls) \mid t \in \text{Tree}(\alpha, \beta)\} \cup \text{Pre}_{\textsc{append}}
\]

or, in a more intuitive representation:

\[
\text{Pre}_{\textsc{preorder}} = \text{preorder}(\text{Tree}(\alpha, \beta) \times U_L) \cup \text{append}(\text{GList} \times \text{GList} \times U_L).
\]

This precondition allows us to concentrate on the relevant input queries, abstracting away from ill-typed information which is present in the least Herbrand model of \textsc{preorder}, such as the unintended atom

\[
\text{preorder}(\text{tree}(0, \text{leaf}([ ]), \text{leaf}(\text{nil})), [0, [ ]], \text{nil})).
\]

Indeed, a complete characterization of \( M^t_{\textsc{preorder}} \) is much more laborious than simply reasoning on correctness and termination of atoms in \( \text{Pre}_{\textsc{preorder}} \).

A candidate level mapping is:

\[
||\text{preorder}(t, Ls)|| = ||t|| + 1 \quad ||\text{append}(\text{xs}, \text{ys}, \text{zs})|| = ||\text{xs}||.
\]

The +1 adjustment is needed to satisfy the required proof obligations, but, as shown later in Section 4.2, the method can be refined to avoid this complication.

Finally, we define the postcondition, which reflects the \textit{intended interpretation} of \textsc{preorder}:
\[ \text{Post}_{\text{PREORDER}} = \{ \text{preorder}(t, ls) \mid t \in \text{Tree}(x, \beta) \land \]

\[ \text{ls is a preorder traversal of } t \} \cup \text{Post}_{\text{APPEND}}. \]

We are now in the position to prove \( \vdash \{ \text{Pre}_{\text{PREORDER}} \} \text{ PREORDER } \{ \text{Post}_{\text{PREORDER}} \} \) by showing the proof obligations of Definition 3.2.

For clause (p1), we have to show that if preorder(nil, []) is in the precondition then it is in the postconditions as well, which is obvious. The reasoning on (p2) and on the clauses of APPEND is also immediate. Let us concentrate on clause (p3), and consider a ground instance:

\begin{align*}
\text{preorder} & (\text{tree}(x, \text{left}, \text{right}), \text{ls}) \leftarrow \\
& \quad \text{preorder} (\text{left}, \text{as}) \\
& \quad \text{preorder} (\text{right}, \text{bs}) \\
& \quad \text{append} ([x | \text{as}], \text{bs}, \text{ls}).
\end{align*}

Assume that the head is in the precondition, i.e. that \( x \in \beta \) and \( \text{left}, \text{right} \in \text{Tree}(x, \beta) \). By definition of \( \text{Pre}_{\text{PREORDER}} \), preorder(left, as) and preorder(right, bs) are also in the precondition. Moreover.

\[ |\text{preorder}(\text{tree}(x, \text{left}, \text{right}), \text{ls})| = |\text{left}| + |\text{right}| + 2 \]
\[ > |\text{left}| + 1 \]
\[ = |\text{preorder}(\text{left}, \text{as})| \]

and analogously for preorder(right, bs). This shows the proof obligation (1) of Definition 3.2 in the cases \( i = 1, 2 \).

In the case \( i = 3 \), we have to show that append([x | as], bs, ls) is in \( \text{Pre}_{\text{PREORDER}} \), i.e. that \([x | \text{as}] \) and bs are ground lists, under the assumption that preorder(left, as) and preorder(right, bs) are in \( \text{Post}_{\text{PREORDER}} \), which is obvious. Moreover,

\[ |\text{preorder}(\text{tree}(x, \text{left}, \text{right}), \text{ls})| = |\text{left}| + |\text{right}| + 2 \]
\[ > |\text{left}| + 1 \]
\[ = |\text{as}| + 1 \]
\[ = |\text{append}([x | \text{as}], \text{bs}, \text{ls})| \]

since preorder(left, as) \( \in \text{Post}_{\text{PREORDER}} \) implies that the number of nodes of left is equal to the length of as.

Finally, consider the proof obligation (2). Under the hypothesis that:

\begin{align*}
\text{tree} & (x, \text{left}, \text{right}) \in \text{Tree}(x, \beta), \text{ and} \\
\text{as} & \text{ is a preorder traversal of } \text{left}, \text{ and} \\
\text{bs} & \text{ is a preorder traversal of } \text{right}, \text{ and} \\
\text{ls} & = [x | \text{as}] \ast \text{bs}.
\end{align*}

we have to show that ls is a preorder traversal of tree(x, left, right), which is immediate.

In conclusion, \( \vdash \{ \text{Pre}_{\text{PREORDER}} \} \text{ preorder}(T, L) \{ \text{Post}_{\text{PREORDER}} \} \) holds. Lifting up to non-ground queries, we observe that

\[ \vdash \{ \text{Pre}_{\text{PREORDER}} \} \text{ preorder}(T, L) \{ \text{Post}_{\text{PREORDER}} \} \]
holds, where $T$ is a (possibly non-ground) term with every ground instance in $\text{Tree}(x, \beta)$, and $L$ is a variable. The query $\text{preorder}(T, L)$ is intended to calculate the preorder traversal of $T$. In particular, if $x = \beta = U$ then the nodes of the tree $T$ can be any term. Analogously, if $x = \text{List}(U)$ the leaves can be any list.

### 3.1.2. Proof outlines

The proof that $\vdash \{\text{Pre}\} \quad P \quad \{\text{Post}\}$ holds can be presented in a suggestive way by means of proof outlines, a well-known tool in verification of imperative programs.

**Definition 3.3.** A proof outline for a clause $A \leftarrow A_1, \ldots, A_n$ and $\{\text{Pre}\}, \{\text{Post}\}$ is a labeled clause of the form:

\[
\begin{align*}
&\{g_0\} \\
&\{f_0\} \\
&\ldots \\
&\{f_{n-1}\} \\
&\{g_n\} \\
&\{f_n\}
\end{align*}
\]

where $t_i$ and $f_i, g_i$, for $i \in \{0, n\}$ are respectively integer expressions and assertions (in some formal logic), such that every ground instance of the following proof obligations is satisfied:

1. for $i \in \{0, n\}$: $t_i = |A_i|$
2. for $i \in \{0, n\}$: $g_i \iff A_i \in \text{Pre}$
3. for $i \in \{0, n\}$: $f_i \iff A_i \in \text{Post}$
4. for $i \in \{0, n\}$: $g_0 \land f_0 \land \ldots \land f_{i-1} \Rightarrow g_{i} \land t_{0} > t_{i}$
5. $g_0 \land f_0 \land \ldots \land f_{n} \Rightarrow f_{n}$

The intuition is that the assertion $g_i$ specifies the conditions under which the atom $A_i$ of the clause is in $\text{Pre}$, the assertion $f_i$ specifies the conditions under which $A_i$ is in $\text{Post}$, and the expressions $t_i$ represent the level of $A_i$. Under this assumption, the proof obligations (iv) and (v) directly reflect respectively Definition 3.2 (1) and (2).

By construction, $\vdash \{\text{Pre}\} \quad P \quad \{\text{Post}\}$ holds if and only if there exists a level mapping $\{\text{Pre}\}$ and a proof outline for each clause of $P$ and $\{\text{Post}\}$. Let us see the proof outlines for $\text{PREORDER}$.

\[
\begin{align*}
&\{\text{nil} \in \text{Tree}(x, \beta)\} \\
&\text{preorder}($\text{nil}, [\ ]$). \quad \{1\} \\
&\{\text{nil} \in \text{Tree}(x, \beta) \land [\ ] \text{ is a preorder traversal of it} \}
\end{align*}
\]

\[
\begin{align*}
&\{\text{leaf}(X) \in \text{Tree}(x, \beta)\} \\
&\text{preorder}($\text{leaf}(X), [X]$). \quad \{2\} \\
&\{\text{leaf}(X) \in \text{Tree}(x, \beta) \land [X] \text{ is a preorder traversal of it} \}
\end{align*}
\]
The related proofs have been shown in the previous section. Proof outlines become simpler when considering the relation $\vdash$. In the labeled clause no $t_i$ appears and the proof obligations are (ii–iii–v) plus the following simplified version of (iv):

$(iv')$ for $i \in [1,n]$: $g_0 \land f_1 \land \ldots \land f_{i-1} \Rightarrow g_i$.

Again, by construction $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds iff there exists a proof outline for each clause of $P$ and $\text{Pre, Post}$. It should be observed how proof outlines represent in a concise way the domino-style propagation of premises in the proof obligations of Definition 3.2.

We also present a more general form of proof outlines, which enable us to simplify proofs by using assertions $f_i$ and $g_i$ which do not coincide with post and pre-conditions, but rather represent strengthenings and weakenings of $\text{Post}$ and $\text{Pre}$, according to the following reformulation of proof outlines. We denote with $h_i$ for $i \in [0,n+1]$ the assertion $g_0 \land f_1 \land \ldots \land f_{i-1}$. Thus, in particular, $h_0 = h_1 = g_0$.

(i) for $i \in [0,n]$: $h_i \land g_i \Rightarrow t_i = |A_i|$

(ii) $g_0 \Leftarrow A_0 \in \text{Pre}$, and for $i \in [1,n]$: $h_i \land g_i \Rightarrow A_i \in \text{Pre}$.

(iii) for $i \in [1,n]$: $f_1 \Leftarrow A_i \in \text{Post} \land h_i$, and $h_{i+1} \land f_0 \Rightarrow A_0 \in \text{Post}$.

(iv) for $i \in [1,n]$: $h_i \Rightarrow g_i \land t_0 > t_i$.

(v) $h_{n+1} \Rightarrow f_0$.

The construction of a proof outline satisfying the proof obligations (i) through (v) for every clause in a program $P$ amounts to showing that $\vdash_r \{\text{Pre}\} P \{\text{Post}\}$ holds. In the following figure it is depicted how (i) through (iv) imply conditions 1(a) and 1(b) of Definition 3.2.

Similarly, the proof obligations (ii,iii,v) imply condition 2 of Definition 3.2.

### 3.2. Modularity

Proving $\vdash \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash_r \{\text{Pre}\} P \{\text{Post}\}$ can be a difficult task when considering either large programs or complex specifications. We will discuss the former case in Section 4.2, by providing some results in order to show that

$$\vdash \{\text{Pre}\} P \cup Q \{\text{Post}\}$$
holds, starting from triples for \( P \) and \( Q \). In this section, we investigate how to prove
\[ \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \]
starting from triples involving simpler specifications. First, we show that the following (Hoeare's logic style) rule is valid:

**Theorem 3.1.**

\[
\begin{align*}
\vdash \{ \text{Pre} \} & \ P \ \{ \text{Post} \} \\
\vdash \{ \text{Pre} \} & \ P \ \{ \text{Post}' \} \\
\vdash \{ \text{Pre} \} & \ P \ \{ \text{Post} \cap \text{Post}' \} 
\end{align*}
\]

**Proof.** Let \( A \leftarrow B_1, \ldots, B_n \) be a ground instance of a clause from \( P \).
- for \( i \in [1..n] \), if \( \text{Pre} \models A \) and \( \text{Post} \cap \text{Post}' \models B_1, \ldots, B_{i-1} \) then \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) implies \( \text{Pre} \models B_i \);
- if \( \text{Pre} \models A \) and \( \text{Post} \cap \text{Post}' \models B_1, \ldots, B_n \) then \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) implies \( \text{Post} \models A \) and \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post}' \} \) implies \( \text{Post}' \models A \). Therefore \( \text{Post} \cap \text{Post}' \models A \). \( \square \)

The importance of this rule is twofold. On the one hand, it is relevant from a practical point of view, since the proof of a triple is split into two simpler proofs. For instance, if we have to show the correctness of a sorting program, with postcondition
\[
\{ \text{sort}(\text{xs}, \text{ys}) \mid \text{ys} \text{ is an ordered permutation of xs} \}
\]
then we can split it into:
\[
\{ \text{sort}(\text{xs}, \text{ys}) \mid \text{ys} \text{ is a permutation of xs} \}
\]
and
\[
\{ \text{sort}(\text{xs}, \text{ys}) \mid \text{ys} \text{ is an ordered list} \}
\]
Proving separately the correctness of the two simpler postconditions generally involves less effort than proving the correctness of their conjunction. On the other hand, Theorem 3.1 allows us to define the notion of *strongest postcondition*.

**Definition 3.4.** Assume that \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) holds. We denote by \( \text{sp}(P, \text{Pre}) \) the intersection of every \( \text{Post}' \) such that \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post}' \} \), and call it the *strongest postcondition* of \( P \) and \( \text{Pre} \).

It is worth noting that the strongest postcondition is defined only for programs for which there exists at least one \( \text{Post} \) such that \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \). In fact, there exist programs and preconditions for which \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) holds for no \( \text{Post} \), such as \( p \leftarrow q \) and \( \text{Pre} = \{ p \} \).

Symmetrically, the following rule holds, which allows us to simplify preconditions.

**Theorem 3.2.**

\[
\begin{align*}
\vdash \{ \text{Pre} \} & \ P \ \{ \text{Post} \} \\
\vdash \{ \text{Pre}' \} & \ P \ \{ \text{Post} \} \\
\vdash \{ \text{Pre} \cup \text{Pre}' \} & \ P \ \{ \text{Post} \} 
\end{align*}
\]

We introduce now the notion of *weakest liberal precondition*, defined as the union of all preconditions.

**Definition 3.5.** We denote by \( \text{wlp}(P, \text{Post}) \) the union of every \( \text{Pre}' \) such that \( \vdash \{ \text{Pre}' \} \ P \ \{ \text{Post} \} \), and call it the *weakest liberal precondition* of \( P \) and \( \text{Post} \).
The next result shows that $sp(P, Pre)$ and $wlp(P, Post)$ are, respectively, a postcondition and a precondition.

**Theorem 3.3.**

$$\vdash \{Pre\} P \{Post\} \quad \vdash \{Pre\} P \{sp(P, Pre)\} \quad \vdash \{wlp(P, Post)\} P \{Post\}.$$

We next extend Theorems 3.1 and 3.2 to relation $\vdash_r$.

**Theorem 3.4.**

$$\vdash_r \{Pre\} P \{Post\} \quad \vdash_r \{Pre\} P \{Post'\} \quad \vdash_r \{Pre\} P \{Post \cap Post'\}.$$

**Proof.** Suppose that $\vdash_r \{Pre\} P \{Post\}$ holds using the level mapping $\|_1$, and $\vdash_r \{Pre\} P \{Post'\}$ holds using $\|_2$. Then we show that $\vdash_r \{Pre\} P \{Post \cap Post'\}$ holds using $\|_1 + \|_2$ (or equivalently $\min\{|_1, |_2\}$). By Theorem 3.1, we have only to show the decreasing of the level mapping. Consider $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$, and $i \in [1, n]$. If $Pre \models A \wedge Post \cap Post' \models B_1, \ldots, B_{i-1}$ then, by hypothesis, $|A_1| > |B_{i-1}|$ and $|A_2| > |B_{i-1}|$. Therefore $|A_1| + |A_2| > |B_{i-1}| + |B_{i-1}|$, and analogously for $\min$.

We point out that it is not necessary to define a notion of strongest terminating postcondition as the intersection of all $Post$ such that $\vdash_r \{Pre\} P \{Post\}$. In fact, $\vdash_r \{Pre\} P \{Post\}$ trivially implies $\vdash_r \{Pre\} P \{sp(P, Pre)\}$, and therefore $sp(P, Pre)$ is the strongest terminating postcondition.

On the other hand, it is important to define a notion of weakest precondition of $P$ and $Post$ as the union of every precondition $Pre$ such that $\vdash_r \{Pre\} P \{Post\}$. We start by stating the following rule:

**Theorem 3.5.**

$$\vdash_r \{Pre\} P \{Post\} \quad \vdash_r \{Pre'\} P \{Post\} \quad \vdash_r \{Pre \cup Pre'\} P \{Post\}.$$

**Proof.** Assume that $\vdash_r \{Pre\} P \{Post\}$ holds using the level mapping $\|_1$, and that $\vdash_r \{Pre'\} P \{Post\}$ holds using the level mapping $\|_2$. By defining $\| A \|$ as follows:

$$\| A \| = \begin{cases} 
\min\{|A_1|, |A_2|\} & \text{if } A \in Pre \cap Pre', \\
|A_1| & \text{if } A \in Pre \setminus Pre', \\
|A_2| & \text{if } A \in Pre' \setminus Pre \\
0 & \text{otherwise}
\end{cases}$$

and $\| A \| = 0$ otherwise, it is readily checked that $\vdash_r \{Pre \cup Pre'\} P \{Post\}$ holds using the level mapping $\| \|$.

As an example, consider again the `APPEND` program. One of its uses is to compute the list obtained by concatenating two given lists. In fact, we have that $\vdash_r \{Pre_{APPEND}\} APPEND \{Post_{APPEND}\}$ holds by the level mapping $\|_1$, where:

$$\| \text{append}(xs, ys, zs) \|_1 = |xs|.$$
In addition to this use (or directionality), APPEND is also employed to extract prefixes or suffixes of a given list. For instance, for a list zs, a computed instance of \texttt{append(xs, ys, zs)} binds xs to a prefix and ys to a suffix of zs. Formally, one can show that \( \vdash \{ \text{Pre} \} \text{APPEND} \{ \text{Post}_{\text{APPEND}} \} \) holds by the level mapping \( | \cdot |_2 \), where

\[
\text{Pre} = \text{append}(U_l \times U_l \times \text{GList})
\]

\[
|\text{append}(xs, ys, zs)|_2 = |zs|.
\]

By Theorem 3.5, we conclude that \( \vdash \{ \text{Pre}_{\text{APPEND}} \cup \text{Pre} \} \text{APPEND} \{ \text{Post}_{\text{APPEND}} \} \) holds by the level mapping:

\[
|\text{append}(xs, ys, zs)| = \begin{cases} 
\min\{|xs|,|zs|\} & \text{if } xs, ys, zs \in \text{GList}, \\
|xs| & \text{if } xs, ys \in \text{GList}, zs \notin \text{GList}, \\
|zs| & \text{if } xs, ys \notin \text{GList}, zs \in \text{GList}, \\
0 & \text{otherwise}.
\end{cases}
\]

The next definition introduces the notion of the weakest precondition.

**Definition 3.6.** We denote by \( \text{wp}(P, \text{Post}) \) the union of every \( \text{Pre}' \) such that \( \vdash \{ \text{Pre}' \} P \{ \text{Post} \} \), and call it the weakest precondition of \( P \) and \( \text{Post} \).

In general, the weakest precondition and the weakest liberal precondition do not coincide. As an example, consider the following program \( P \)

\[
P = p.
\]

and \( \text{Post} = \{ p \} \). We have that

\[
\text{wp}(P, \text{Post}) = \{ p \} \neq \emptyset = \text{wp}(P, \text{Post}).
\]

By generalizing the proof of Theorem 3.5, we obtain that \( \text{wp}(P, \text{Post}) \) is indeed a precondition for \( P \) and \( \text{Post} \).

**Theorem 3.6.**

\[
\vdash \{ \text{wp}(P, \text{Post}) \} P \{ \text{Post} \}.
\]

**Proof.** Consider the sequence \( \text{Pre}_1, \text{Pre}_2, \ldots \) of all preconditions \( \text{Pre}_i \) such that \( \vdash \{ \text{Pre}_i \} P \{ \text{Post} \} \) holds using \( | \cdot |_\cdot \). We define \( \|A\|^i = \min \{ |A| : A \in \text{Pre}_i, i \geq 1 \} \), for \( A \in \text{wp}(P, \text{Post}) \), and \( \|A\|^i = 0 \) otherwise. We show that \( \vdash \{ \text{wp}(P, \text{Post}) \} P \{ \text{Post} \} \) holds using \( \| \cdot \| \). Consider \( A = B_1, \ldots, B_n \in \text{ground}_i(P) \).

- **Assume that, for \( i \in [1..n], \text{wp}(P, \text{Post}) \models A \) and \( \text{Post} \models B_1, \ldots, B_{i-1} \). For every \( \text{Pre}_i \) such that \( A \in \text{Pre}_i \), we have that \( \text{Pre}_i \models B_i \) and \( |A|^i > |B_i|^i \). Therefore, \( \text{wp}(P, \text{Post}) \models B_i \) and \( \|A|^i = \min \{ |A|^i : A \in \text{Pre}_i, i \geq 1 \} > \min \{ |B_i|^i : B_i \in \text{Pre}_i, i \geq 1 \} = \|B_i|^i \).**

- **If \( \text{wp}(P, \text{Post}) \models A \) and \( \text{Post} \models B_1, \ldots, B_n \), then there exists \( \text{Pre}_i \) such that \( \text{Pre}_i \models A \) and \( \text{Post} \models B_1, \ldots, B_n \). Therefore, \( \text{Post} \models A \). \( \Box \)**

We conclude this section by introducing two further rules.
Theorem 3.7.

\[ \vdash \{ \text{Pre} \} P \{ \text{Post} \} \quad \vdash \{ \text{Pre} \} P \{ \text{Post} \cap \text{Pre} \} \quad \vdash \{ \text{Pre} \} P \{ \text{Post} \cap \text{Pre} \} . \]

We already observed that in Definition 3.2, the values assumed by a level mapping are relevant only on the set \( \text{Pre} \). For postconditions, the above theorem shows that \( \text{Post} \cap \text{Pre} \) is a postcondition if \( \text{Post} \) is so. Intuitively, this fact is justified by the intuition that a postcondition describes a property of atoms in \( \text{Pre} \) which are consequences of the program.

The next sections are devoted to study the relation of \( \vdash \) and \( \vdash \) with the four introduced notions of correctness, namely (weak) partial and (weak) total.

### 3.3. Call patterns and weak partial correctness

A key result is the following theorem, stating the persistency of the relation \( \vdash \) along SLD-derivations of programs \( P \) and queries \( Q \).

**Theorem 3.8.** (Persistency) Assume that

\[ \vdash \{ \text{Pre} \} P \{ \text{Post} \} \quad \text{and} \quad \vdash \{ \text{Pre} \} Q \{ \text{Post} \}. \]

Then for every SLD-resolvent \( Q' \) of \( P \) and \( Q \), \( \vdash \{ \text{Pre} \} Q' \{ \text{Post} \} \) holds.

**Proof.** First, we show that if \( Q = A_1, \ldots, A_n \) is a ground query and

\[ A_i \leftarrow B_1, \ldots, B_l \in \text{ground}_l(P) \]

then the ground resolvent

\[ Q' = A_1, \ldots, A_{i-1}, B_1, \ldots, B_l, A_{i+1}, \ldots, A_n \]

is such that \( \vdash \{ \text{Pre} \} Q' \{ \text{Post} \} \) holds. Let us show the proof obligations.

- For \( k \in [1, i-1] \), if \( \text{Post} \models A_1, \ldots, A_{i-1} \) then, since \( \vdash \{ \text{Pre} \} Q \{ \text{Post} \} \), \( \text{Pre} \models A_i \).

- For \( k \in [1, l] \), if \( \text{Post} \models A_1, \ldots, A_{i-1}, B_1, \ldots, B_l \) then \( \text{Pre} \models A_i \), since \( \text{Post} \models A_1, \ldots, A_{i-1} \) and \( \vdash \{ \text{Pre} \} Q \{ \text{Post} \} \) holds. In addition, \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) and \( \text{Pre} \models A_i \land \text{Post} \models B_1, \ldots, B_l \) imply \( \text{Pre} \models B_k \).

- For \( k \in [i+1, n] \), if \( \text{Post} \models A_1, \ldots, A_{i-1}, B_1, \ldots, B_l, A_i, \ldots, A_{k-1} \) then \( \text{Pre} \models A_i \), since \( \text{Post} \models A_1, \ldots, A_{i-1} \) and \( \vdash \{ \text{Pre} \} Q \{ \text{Post} \} \) holds. In addition, \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) and \( \text{Pre} \models A_i \land \text{Post} \models B_1, \ldots, B_l \) imply \( \text{Post} \models A_{k-1} \), and then

\[ \text{Post} \models A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_k. \]

Since \( \vdash \{ \text{Pre} \} Q \{ \text{Post} \} \), we conclude \( \text{Pre} \models A_k \).

Consider now a not necessarily ground query \( Q = A_1, \ldots, A_n \), and let

\[ Q' = (A_1, \ldots, A_{i-1}, B_1, \ldots, B_l, A_{i+1}, \ldots, A_n)\theta \]

be the SLD-resolvent of \( Q \) and a variant \( B \leftarrow B_1, \ldots, B_l \) of a clause from \( P \), where \( \theta \) is the mgu of \( A_i \) and \( B \). We point out that every ground instance of \( Q' \) is a ground resolvent of a ground instance of \( Q\theta \) and a ground instance of \( (B \leftarrow B_1, \ldots, B_l)\theta \). Since \( \vdash \{ \text{Pre} \} Q \{ \text{Post} \} \) holds, then \( \vdash \{ \text{Pre} \} Q\theta \{ \text{Post} \} \) holds. From the first part of this proof, we conclude that for every ground instance \( Q'' \) of \( Q' \), \( \vdash \{ \text{Pre} \} Q'' \{ \text{Post} \} \) holds. By Definition 3.2, this is equivalent to say that \( \vdash \{ \text{Pre} \} Q'' \{ \text{Post} \} \) holds, hence the conclusion of the theorem. \( \square \)

Directly from persistency and Definition 3.2 (3), we have that the leftmost atom \( A \) of any query in a SLD-derivation for \( P \) and \( Q \) is true in \( \text{Pre} \), i.e. \( \text{Pre} \models A \). In other
words, \( \text{Pre} \) declaratively characterizes call patterns w.r.t. LD-resolution. Another immediate consequence is that every computed instance of \( P \) and \( Q \) is true in the interpretation \( \text{Post} \). This is a formal counterpart to the intuitive notion that a postcondition is a description of correct instances of intended queries.

**Corollary 3.1.** Assume that \( \models \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) and \( \models \{ \text{Pre} \} \ Q \ \{ \text{Post} \} \). Then for every atom \( A \) selected in a LD-derivation for \( P \) and \( Q \), \( \text{Pre} \models A \).

(i) (Call Patterns) for every atom \( A \) selected in a LD-derivation for \( P \) and \( Q \), \( \text{Pre} \models A \).

(ii) (Computed Instances) for every computed instance \( Q' \) of \( P \) and \( Q \), \( \text{Post} \models Q' \).

**Proof.** (i) Let \( A, Q' \) be a query in a LD-derivation for \( P \) and \( Q \). By Theorem 3.8, we have that \( \models \{ \text{Pre} \} \ A \ \{ \text{Post} \} \). By Definition 3.2, \( \text{Pre} \models A' \) for every instance \( A' \) of \( A \). Therefore \( \text{Pre} \models A \).

(ii) Let \( x_1, \ldots, x_n \) be the variables of \( Q \), and \( p \) be a fresh predicate symbol of arity \( n \). We define:

\[
\text{Pre}' = \text{Pre} \cup \{ p(t_1, \ldots, t_n) \ | \ \text{Post} \models Q(x_i/t_i \ | \ i \in [1..n]) \},
\]

\[
\text{Post}' = \text{Post} \cup p(U_1 \times \ldots \times U_n).
\]

With these assumptions, it is readily checked that \( \models \{ \text{Pre}' \} \ P \ \{ \text{Post}' \} \), since \( p \) does not appear in \( P \), and \( \models \{ \text{Pre}' \} \ Q \), \( p(x_1, \ldots, x_n) \ {\text{Post}'} \), by definition of \( \text{Post}' \). By Strong Completeness of SLD-resolution, there exists a LD-refutation for \( P \) and \( Q \) with computed instance (a variant of) \( Q' \). As a consequence, there exists a LD-derivation for \( P \) and \( Q \), \( p(x_1, \ldots, x_n) \) where \( p(T_1, \ldots, T_n) \) is selected, with \( Q(x_i/T_i \ | \ i \in [1..n]) \) variant of \( Q' \). By (i), \( \text{Pre}' \models p(T_1, \ldots, T_n) \). By definition of \( \text{Pre}' \), we conclude that \( \text{Post} \models Q(x_i/T_i \ | \ i \in [1..n]) \), and then \( \text{Post} \models Q' \). □

Corollary 3.1 is the key result to reason about call patterns and computed/correct instances, as, under the hypothesis of the corollary, \( \text{Pre} \) describes the shapes of every atom selected during a LD-derivation, and \( \text{Post} \) describes a property of computed/correct instances.

As an example, consider a term \( T \) such that every ground instance of it is in \( \text{Tree}(x, \beta) \). We have that \( \models \{ \text{Pre}_{\text{PREORDER}} \} \ \text{preorder}(T, X) \ \{ \text{Post}_{\text{PREORDER}} \} \) holds. By Corollary 3.1, every atom \( \text{preorder}(T', X) \) selected in a LD-derivation of \( \text{PREORDER} \) and \( \text{preorder}(T, X) \) is true in \( \text{Pre}_{\text{PREORDER}} \), which implies that every ground instance of \( T' \) is in \( \text{Tree}(x, \beta) \). In addition, every computed instance \( \text{preorder}(T, X) \) of \( \text{PREORDER} \) and \( \text{preorder}(T, X) \) is true in \( \text{Post}_{\text{PREORDER}} \). In other words, for every ground instance \( \text{preorder}(t, x) \) of \( \text{preorder}(T, X) \), \( x \) is a preorder traversal of \( t \). In general, proof relation \( \models \) is sound for proving weak partial correctness.

**Theorem 3.9** (Weak partial correctness). If \( \models \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) holds then \( P \) is weak partially correct w.r.t. the specification \( (\text{Pre}, \text{Post}) \), i.e.

\[
M_P^t \cap \text{Pre} \subseteq \text{Post}.
\]

**Proof.** Consider \( A \in M_P^t \cap \text{Pre} \). Then \( \models \{ \text{Pre} \} \ A \ \{ \text{Post} \} \) holds, and by Strong Completeness of SLD-resolution there exists a LD-refutation for \( P \) and \( A \). Therefore \( A \in \text{Post} \) by Corollary 3 (ii). □
Notice, however, that the proof method is not complete with respect to the proposed definition of weak partial correctness. As an example, if we consider \( P \) as:
\[
p \leftarrow q.
\]
and \( \text{Post} = \{ \emptyset \} \), we have that \( P \) is weak partially correct w.r.t. the specification \((\text{Pre}, \text{Post})\) but \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) does not hold for any \( \text{Post}' \). The reason lies in the fact that the proof method based on relation \( \vdash \) addresses call pattern characterization as well as correctness, in the sense clarified by Corollary 3.1 (i). In the example, \( \text{Pre} \) does not characterize call patterns of \( P \) and \( p \) since \( \text{Pre} \not\nvdash q \).

### 3.4. Partial correctness

The Weak Partial Correctness Theorem 3.9 suggests to investigate further the relation between postconditions and the well-typed fragment of the least Herbrand model, i.e. \( M_p^I \cap \text{Pre} \). Since every postcondition is a superset of \( M_p^I \cap \text{Pre} \), if we prove that \( \vdash \{ \text{Pre} \} P \{ M_p^I \cap \text{Pre} \} \), then we conclude that \( M_p^I \cap \text{Pre} \) is the strongest postcondition. This fact is made precise in the following theorem.

**Theorem 3.10.** Assume that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds.

Then \( \vdash \{ \text{Pre} \} P \{ M_p^I \cap \text{Pre} \} \) holds, and then
\[
\text{sp}(P, \text{Pre}) = M_p^I \cap \text{Pre}.
\]

**Proof.** Consider \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P) \).
- For \( i \in [1, n] \), if \( \text{Pre} \models A \land M_p^I \cap \text{Pre} \models B_1, \ldots, B_{i-1} \) then by Theorem 3.9, \( \text{Post} \models B_1, \ldots, B_{i-1} \), and, by the hypothesis, \( \text{Pre} \models B_i \).
- Suppose that \( \text{Pre} \models A \land M_p^I \cap \text{Pre} \models B_1, \ldots, B_n \). As \( M_p^I \) is a model of \( P \), we conclude that \( M_p^I \models A \), and then \( M_p^I \cap \text{Pre} \models A \).

Therefore \( \vdash \{ \text{Pre} \} \rightarrow \{ M_p^I \cap \text{Pre} \} \) holds. By Definition 3.4 we have that \( \text{sp}(P, \text{Pre}) \subseteq M_p^I \cap \text{Pre} \). The converse inclusion \( \text{sp}(P, \text{Pre}) \supseteq M_p^I \cap \text{Pre} \) follows from Theorems 3.3 and 3.9. \( \square \)

As a consequence, if \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds, then \( P \) is partially correct w.r.t. the specification \((\text{Pre}, \text{sp}(P, \text{Pre}))\).

**Theorem 3.11** (Partial correctness). If \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds then \( P \) is partially correct w.r.t. the specification \((\text{Pre}, \text{sp}(P, \text{Pre}))\).

The problem is now how to characterize the strongest postcondition directly, without first constructing the least Herbrand model \( M_p^I \). To this end, we define the notion of well-supported interpretation, introduced in the context of semantics for general logic programs (see Ref. [2]). A similar notion has been employed for program verification w.r.t. three-valued semantics by Malfon [42].

**Definition 3.7.** Assume that \( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds.

\( \text{Post} \) is a well-supported interpretation (w.r.t. \( P \) and \( \text{Pre} \)) iff there exist:
- a well-founded poset \((W, \prec)\), and
- a function \( \vdash : B_L \rightarrow W \)
such that for every $A \in \text{Post} \cap \text{Pre}$ there exists $A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)$ such that: $\forall i \in [1,n]: \text{Post} \models B_i \land |A| > |B_i|.$

Observe that the condition $\text{Post} \models B_i$ for $i \in [1,n]$ is equivalent to $\text{Post} \cap \text{Pre} \models B_i$ for $i \in [1,n]$ under the assumption that $\vdash \{\text{Pre}\} \ P \ {\text{Post}}$ holds. The idea underlying this definition is to require that every atom in $\text{Post} \cap \text{Pre}$ has a successful ground derivation. In fact, for any such atom there exists a ground derivation which is finite, because the poset is well-founded, and successful, because the last selected atom unifies with some clause head. Therefore, a well-supported postcondition is a subset of the interesting fragment of the least Herbrand model, i.e. well-supportedness combined with the proof relation $\vdash$ is a proof method for partial correctness.

**Theorem 3.12.** Let $P$ be a program such that $\vdash \{\text{Pre}\} \ P \ {\text{Post}}$. Then

\[
\text{Post} \cap \text{Pre} = \operatorname{sp}(P, \text{Pre}) \iff \text{Post} \text{ is well-supported (w.r.t. } P \text{ and } \text{Pre}).
\]

**Proof.** (only-if) We consider natural numbers with the usual ordering relation $<$ as the well-founded poset. Let $| | : B_L \to N$ be the function:

\[
|A| = \begin{cases} 
\min\{i : A \in T_P \uparrow i\} & \text{if } A \in M^f_L \cap \text{Pre} \\
0 & \text{otherwise.}
\end{cases}
\]

$| |$ is well-defined since $M^f_L = T_P \uparrow \omega$. By hypothesis, $\text{Post} \cap \text{Pre} = \operatorname{sp}(P, \text{Pre})$, and then, by Theorem 3.10, $\text{Post} \cap \text{Pre} = M^f_L \cap \text{Pre} = T_P \uparrow \omega \cap \text{Pre}$. To show the conclusion, we prove by induction on $i \geq 0$ that for every $A \in T_P \uparrow i \cap \text{Pre}$ there exists $A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)$ such that:

\[
\forall i \in [1,n]: \text{Post} \models B_i \land |A| > |B_i|.
\]

The base case is trivial, since $T_P \uparrow 0 = \emptyset$. Let $A$ be in $T_P \uparrow i \cap \text{Pre}$. If $A \in T_P \uparrow (i - 1)$ then the conclusion follows from the inductive hypothesis. On the contrary, let $A$ be in $T_P \uparrow i \setminus T_P \uparrow (i - 1) \cap \text{Pre}$. By definition of $T_P$, there exists $A \leftarrow B_1, \ldots, B_n$ in $\text{ground}_L(P)$ such that

\[
T_P \uparrow (i - 1) \models B_1, \ldots, B_n.
\]

(1)

We now observe that, by Theorem 3.9,

\[
\text{Post} \supseteq M^f_L \cap \text{Pre} \supseteq T_P \uparrow (i - 1) \cap \text{Pre}.
\]

Since $\vdash \{\text{Pre}\} \ P \ {\text{Post}}$ hold, by a simple induction on $n$, this and (1) imply

\[
\text{Post} \models B_1, \ldots, B_n.
\]

In addition, $A \in T_P \uparrow i \setminus T_P \uparrow (i - 1)$ implies that for $k \in [1,n]$

\[
|A| = i > i - 1 \geq |B_k|.
\]

(if) We only show that $\text{Post} \cap \text{Pre} \subseteq M^f_L$, as the converse inclusion follows directly from Theorem 3.9. Consider $A \in \text{Post} \cap \text{Pre}$ and a maximal tree $T$ such that:

- $A$ is the root;
- if $B$ is a node such that $B \in \text{Post} \cap \text{Pre}$ and $B_1, \ldots, B_n$ are its children then $B \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)$ and

\[
\text{Post} \models B_1, \ldots, B_n \land |B| > \max_{i=1}^n |B_i|.
\]
Since $W$ is well-founded, there is no infinite branch: by the König lemma, the tree is finite. Since $\vdash \{Pre\} P \{Post\}$ and $A \in Pre$, it is readily checked by induction that every atom in $T$ is in $Post \cap Pre$. If a leaf $B$ is not a ground instance of a fact in $P$, then, by hypothesis, there exists $B \leftarrow B_1, \ldots, B_n$ in $ground_e(P)$ such that $Post \models B_1, \ldots, B_n$ and $|B| > \max_{i=1}^n |B_i|$, thus $T$ is not maximal. In conclusion, $T$ is a proof tree for $A$ (see [22]), which implies $A \in M_f$.

As a final observation, we point out that when $\vdash \{Pre\} P \{Post\}$ holds, then $sp(P, Pre)$ is well-supported. In fact, by Theorem 3.10 $\vdash \{Pre\} P \{sp(P, Pre)\}$ holds, and $sp(P, Pre) \cap Pre = sp(P, Pre)$. Thus, we are in the hypothesis of Theorem 3.12.

### 3.4.1. Proof outlines

Proving weak partial correctness is handy, as the task can be carried out using the proof outlines. In contrast, Definition 3.7 may seem intricate and difficult to handle. Fortunately, it has a straightforward interpretation in terms of proof outlines.

**Definition 3.8.** A proof outline for a clause $A \leftarrow A_1, \ldots, A_n$, a function $| | : BL \rightarrow W$ into a well-founded poset $(W, <)$ and $Pre, Post$, is a labeled clause of the form:

$$
\begin{align*}
\{g\} & \\
A_0 \leftarrow & \{t_0\} \\
A_1, \leftarrow & \{t_1\} \\
\{f_1\} & \\
\vdots & \\
A_{n-1}, \leftarrow & \{t_{n-1}\} \\
\{f_{n-1}\} & \\
A_n, \leftarrow & \{t_n\} \\
\{f_n\} &
\end{align*}
$$

where $t_i$ for $i \in [0, n]$ and $f_i, g$, for $i \in [1, n]$ are, respectively, expressions over $W$ and assertions (in some formal logic), such that every ground instance of the following proof obligations holds:

(i) for $i \in [0, n]$: $g \Rightarrow t_i = |A_i|$, 
(ii) for $i \in [1, n]$: $g \land f_i \Rightarrow A_i \in Post$, 
(iii) for $i \in [1, n]$: $g \Rightarrow f_i \land t_0 > t_i$.

The assertion $g$ expresses a relation among the variables of the clause in such a way that for every ground instance of the proof outline if the instance of $g$ holds then $A_i \in Post \land |A_0| > |A_i|$. The use of the auxiliary assertions $f_i$'s is not strictly necessary, albeit useful in constructing proof outlines.

Next, in order to prove that $Post$ is well-supported w.r.t. $P$ and $\bar{r}_$. we have to show that there exist proof outlines for (instances of) clauses from $P$ and a function $| | : BL \rightarrow W$ such that every atom $A$ in $Post \cap Pre$ is a ground instance of some head atom in a proof outline and the assertion associated with the head holds.
3.4.2. Example: lexicographic ordering

As an example, consider the following program LEXORD, specifying a lexicographic ordering relation over pairs of natural numbers.

\[(/1) \leq ([X, Y], [X, Y])\]

\[(/2) \leq ([X, Y], [s(U), V]) \leftarrow \leq ([X, Y], [U, Y])\]

\[(/3) \leq ([X, Y], [s(V)]) \leftarrow \leq ([X, Y], [V])\]

For notational convenience, we shall identify the natural number \(a\) with the term \(s^a(0)\). It is readily checked that \(\{\text{Pre}\} \text{LEXORD} \{\text{Post}\}\) holds, where

\[\text{Pre} = \{ \leq ([x, y], [u, v]) \mid x, y, u, v \in N \}\]

\[\text{Post} = \{ \leq ([x, y], [u, v]) \mid x, y, u, v \in N, (x, y) \leq \kappa_v(u, v) \}\]

where \(\leq \kappa_v\) is the lexicographic ordering relation. We observe that one would have expected clauses \((/2, /3)\) to be more general. In fact, a natural version of the program is the following:

\[(/4) \leq ([X, Y], [s(U), V]) \leftarrow \leq ([X, Y], [U, Z])\]

\[(/5) \leq ([X, Y], [U, s(V)]) \leftarrow \leq ([X, Y], [U, V])\]

Although \((/2, /3)\) are instances of \((/4, /5)\), we are still in the position to show that LEXORD is partially correct w.r.t. the specification \(\{\text{Pre, Post}\}\), by means of Theorem 3.12. We fix \(W = (N \times N, \leq \kappa_v)\). and define \(|\ |\) as follows:

\[|\leq ([x, y], [u, v])| = (u - x - v - y)\]

where difference over natural numbers is interpreted as a total function, by letting for \(a, b \in N, a - b = 0\) if \(a < b\). The proof outlines for well-supportedness follow.

\[(p1) \{ X, Y \in N \} \leq ([X, Y], [X, \_]) \{ (0, 0) \}\]

\[(p2) \{ X, Y, U, V \in N \land X < s(U) \} \leq ([X, Y], [s(U), V]) \leftarrow \leq ([X, Y], [U, Y]) \{ (U - X + 1, V - Y) \}\]

\[(p3) \{ X, Y, V \in N \land Y < s(U) \} \leq ([X, Y], [X, s(V)]) \leftarrow \leq ([X, Y], [V]) \{ (0, V - Y + 1) \}\]

Since the decreasing of \(|\ |\) is obvious, we need only to show that every atom in \(\text{Pre} \cap \text{Post}\) is an instance of some head, and that the associated assertion holds. Let \(\leq ([x, y], [u, v])\) in \(\text{Pre} \cap \text{Post}\). Then \((x, y) \leq \kappa_v(u, v)\).
By definition of \( \leq_{\text{LEXORD}} \), we have that \((x, y) \leq_{\text{LEXORD}} (u, v)\) iff \((x = u \land y = v) \lor (x < u) \lor (x = u \land y < v)\). Proof outlines (p1), (p2) and (p3) cover respectively \((x = u \land y = v)\), \((x < u)\) and \((x = u \land y < v)\). Summarizing, \(\text{Post}\) is well-supported, and, then, partial correctness follows from Theorems 3.12 and 3.11, i.e. \(\text{Post} = \text{Post} \cap \text{Pre} = \text{sp(LEXORD, Pre)} = M_{\text{LEXORD}}^I \cap \text{Pre}\).

3.5. Correct and computed instances

So far, we stressed the fact that postconditions describe correct and computed instances of the intended queries, as formally stated by Corollary 3.1 (ii). However, under certain rather general assumptions, the proposed proof method can be also employed to achieve a full characterization of correct and computed instances of queries. In this section, we present two methods, applicable in different situations.

Given a set of atoms \(I\), we define \(\text{Min}(I) = \{A \in I \mid \exists B \in I : B < A\}\), where \(<\) is the instantiation ordering. We write \(B \leq A\) iff \(A\) is an instance of \(B\); and \(B < A\) iff \(B \leq A\) and not \(A \leq B\). i.e. if \(A\) is an instance of \(B\) which is not a variant of \(B\). We recall that \(\Sigma_L\) is the set of function symbols of the underlying language \(L\).

**Theorem 3.13.** Assume that \(\vdash \{\text{Pre}\} P \{\text{Post}\}\) and \(\vdash \{\text{Pre}\} Q \{\text{Post}\}\) hold with \(\text{Post}\) well-supported, and \(\Sigma_L\) contains infinitely many symbols. Calling \(\mathcal{J} = \{Q_0 \mid \text{Post} \models Q_0\}\), we have that:

(i) \(\mathcal{J}\) is the set of correct instances of \(Q\) w.r.t. \(P\), and

(ii) each of the following conditions is sufficient to state that \(\mathcal{J}\) is the set of computed instance of \(P\) and \(Q\):

(a) \(\text{Min}(\mathcal{J}) = \mathcal{J}\),

(b) \(\mathcal{J}\) is a set of ground queries,

(c) \(\mathcal{J}\) is a finite set.

**Proof.** (i) Since \(\text{Post}\) is well-supported, \(\text{Post} \models Q_0\) iff \(\text{sp}(P, \text{Pre}) \models Q_0\) iff \(M_P^I \cap \text{Pre} \models Q_0\). If \(P \models Q_0\) then \(M_P^I \cap \text{Pre} \models Q_0\), and by the equivalences above, \(Q_0 \in \mathcal{J}\). On the other hand, if \(\text{Post} \models Q_0\) then \(M_P^I \models Q_0\). Since \(\Sigma_L\) is infinite, \(M_P^I \models Q_0 I\), where \(I\) maps every variable of \(Q_0\) into a distinct ground term with functor not appearing in \(P\) or \(Q_0\). \(M_P^I \models Q_0 I\) implies \(P \models Q_0 I\), and by the Theorem on Constants (see Ref. [52]), \(P \models Q_0\).

(ii)(a) Let \(\mathcal{J}'\) be the set of computed instances of \(P\) and \(Q\). By Theorem 3.1(ii), \(\mathcal{J}' \subseteq \mathcal{J}\). Conversely, consider \(Q' \in \mathcal{J}\). By (i) \(P \models Q'\). Therefore, by completeness of LD-resolution there exists \(Q'' \in \mathcal{J}'\) more general than \(Q'\). As pointed out above \(Q'' \in \mathcal{J}\). This and the hypothesis \(\text{Min}(\mathcal{J}) = \mathcal{J}\) imply that \(Q'\) is a variant of \(Q''\). As \(\mathcal{J}\) is closed under the variant-relation, we conclude \(Q' \in \mathcal{J}',\) and then \(\mathcal{J}' = \mathcal{J}\).

(ii)(b) We observe that the hypothesis (ii)(a) holds. Consider \(Q'\) and \(Q''\) in \(\mathcal{J}\) such that \(Q' \leq Q''\). Since \(\mathcal{J}\) is a set of ground queries, \(Q'\) and \(Q''\) are ground and then \(Q' \leq Q''\) implies \(Q' = Q''\). Therefore, for no \(Q', Q'' \in \mathcal{J}\) we have \(Q' < Q''\), and then \(\text{Min}(\mathcal{J}) = \mathcal{J}\).

(ii)(c) Let us show by contrapositive that (ii)(c) \(\Rightarrow\) (ii)(b). If \(Q' \in \mathcal{J}\) is not ground then \(\text{Post} \models Q'\) implies \(\text{Post} \models Q''\) for every ground instance \(Q''\) of \(Q'\). Since \(\Sigma_L\) is infinite, there are infinitely many ground instances of \(Q'\), and then \(\mathcal{J}\) is infinite. \(\square\)
In particular, for atomic queries \( Q \), hypothesis (ii)(b) can be rewritten as \( \mathcal{J} \subseteq Post \). As an example, consider the program \( \text{EVEN} \)

\[
\begin{align*}
\text{even}(0). \\
\text{even}(s(s(X))) & \leftarrow \text{even}(X).
\end{align*}
\]

(defined on a language with \( \Sigma_L \) infinite) and the query \( \text{even}(X) \) with \( Pre = B_1 \), and \( Post = \{ \text{even}(s^k(0)) \mid k \geq 0 \} \). It is easy to show that \( \vdash \{ Pre \} \text{EVEN} \{ Post \} \) holds, and that \( Post \) is well-supported. Also, we have that \( \mathcal{J} = \{ \text{even}(T) \mid Post \models \text{even}(T) \} \) coincides with \( Post \), and then it is a set of ground queries. By Theorem 3.13 (ii)(b), we have that \( \mathcal{J} \) is the set of computed instances of \( \text{EVEN} \) and \( \text{even}(X) \).

Recently, Apt et al. [9] introduced a method for characterizing the computed instances of queries on the basis of the declarative semantics. The method is developed with reference to the notion of partial correctness presented in this paper. The key notion is that of \( (Pre, Post) \)-redundancy-free programs. Roughly, such programs have the property of delivering non-redundant computed instances of the intended queries.

The next result, a slight improvement over [9], shows a sufficient condition that allows us to retrieve the computed instances of the intended queries from \( Post \).

**Theorem 3.14.** Assume that \( \vdash \{ Pre \} \ P \ \{ Post \} \) and \( \vdash \{ Pre \} \ Q \ \{ Post \} \) hold with \( Post \) well-supported, and \( \Sigma_L \) contains infinitely many symbols. If the following conditions hold:

- \textbf{SEM1.} If \( H \leftarrow B_1, \ldots , B_n \) and \( H \leftarrow C_1, \ldots , C_k \) are ground instances of two different clauses in \( P \) then:
  
  \[
  Post \cap Pre \not\subseteq H \wedge B_1, \ldots , B_n \wedge C_1, \ldots , C_k.
  \]

- \textbf{SEM2.} If \( H \leftarrow B_1, \ldots , B_n \) and \( H \leftarrow C_1, \ldots , C_n \) are distinct ground instances of the same clause in \( P \) then:
  
  \[
  Post \cap Pre \not\subseteq H \wedge B_1, \ldots , B_n \wedge C_1, \ldots , C_n.
  \]

then the set of computed instances of \( P \) and \( Q \) is \( \text{Min}\{Q_0 \mid Post \models Q_0\} \).

**Proof.** By Theorem 3.9, \( M_P \cap Pre \not\subseteq Post \cap Pre \). Therefore \( Post \cap Pre \not\subseteq H \wedge B_1, \ldots , B_n \wedge C_1, \ldots , C_k \) implies

\[
M_P \cap Pre \not\subseteq H \wedge B_1, \ldots , B_n \wedge C_1, \ldots , C_k.
\]

Therefore SEM1 and SEM2 imply, respectively, the two conditions assumed in Theorem 7.4 of Ref. [9] to prove that \( P \) is \( (Pre, Post) \)-redundant-free. Corollary 7.2 (iii) from Ref. [9] states that under the hypothesis that \( \Sigma_L \) contains infinitely many symbols, and \( P \) is \( (Pre, Post) \)-redundant-free, the set of computed instances of \( P \) and \( Q \) is \( \text{Min}\{Q_0 \mid M_P \cap Pre \models Q_0\} \). Our conclusion follows by considering that \( M_P \cap Pre \models Q_0 \) iff \( Post \cap Pre \models Q_0 \), since \( Post \) is well-supported (see Theorems 3.12 and 3.10), and \( Post \cap Pre \models Q_0 \) iff \( Post \models Q_0 \), since \( \vdash \{ Pre \} \ Q \ \{ Post \} \) holds. \( \square \)

In certain situations the conditions of Theorem 3.14 can be ensured by means of syntactic restrictions. Namely, condition SEM1 is obviously implied by condition:

- **SYN1.** If \( H_1 \leftarrow B_1, \ldots , B_n \) and \( H_2 \leftarrow C_1, \ldots , C_k \) are variable disjoint variants of different clauses in \( P \), then \( H_1 \) and \( H_2 \) do not unify.

and condition SEM2 is automatically satisfied when:
SYN2. If \( H \leftarrow B_1, \ldots, B_n \in P \), then \( \text{Var}(B_1, \ldots, B_n) \subseteq \text{Var}(H) \)
where \( \text{Var}(X) \) is the set of logic variables appearing in \( X \). As an example, the
program EVEN satisfies both SYN1 and SYN2. We refer the reader to [9] for
several examples and a comparison of the approach with others related to \( \mathcal{F} \)-semantics.

3.6. Weak total correctness

While proof relation \( \vdash \) allows us to reason on (weak) partial correctness, the
additional proof obligations of relation \( \vdash_{r} \) are intended to address universal ter-
mination via the leftmost selection rule, and then (weak) total correctness. On
the other hand, the approach of restricting attention to the atoms in precondi-
tions may facilitate the termination proofs, as will be pointed out in the related
work section.

Our commitment to the study of the leftmost selection rule is made apparent in
the left-to-right propagation of assumptions in proof obligations of Definition 3.2.
However, it should be observed that no assumption is made on the search strategy.
These remarks are formalized by the following results on persistency of \( \vdash_{r} \), and ter-
mination.

**Theorem 3.15 (Persistency and termination).** Assume that \( \vdash_{r} \{\text{Pre}\} P \{\text{Post}\} \) and
\( \vdash_{r} \{\text{Pre}\} Q \{\text{Post}\} \) hold by the same level mapping. Then for every LD-resolvent \( Q' \) of \( P \)
and \( Q \).

\[ \vdash_{r} \{\text{Pre}\} Q' \{\text{Post}\} \] holds by the same level mapping.
Moreover, every LD-tree of \( P \) and \( Q \) is finite.

**Proof.** The fact that \( \vdash_{r} \{\text{Pre}\} Q' \{\text{Post}\} \) holds is shown in Appendix A. Lemma A.1 (iii).
Moreover, by Lemma A.1 (ii) there cannot be an infinite branch in a LD-tree for \( P \) and
\( Q \). Since LD-trees are finitely branching, by König's Lemma, they are finite. \( \square \)

An immediate consequence of this result and Theorem 3.9 is the weak total cor-
rectness theorem.

**Theorem 3.16 (Weak total correctness).** If \( \vdash_{r} \{\text{Pre}\} P \{\text{Post}\} \) holds then \( P \) is weak
totally correct w.r.t. the specification. \( \{\text{Pre}. \text{Post}\} \).

As an example, since \( \vdash_{r} \{\text{Pre}_{\text{PREORDER}}\} \text{PREORDER} \{\text{Post}_{\text{PREORDER}}\} \) holds, then \( \text{PRE-
ORDER} \) is weak totally correct w.r.t. \( \{\text{Pre}_{\text{PREORDER}}, \text{Post}_{\text{PREORDER}}\} \). In addition, for every
term \( T \) whose ground instances are in \( \text{Tree}(x, \beta) \), the LD-tree of \( \text{PREORDER} \) and
\( \text{preorder}(T, L) \) is finite.

3.7. Total correctness

First, we state the analogous of Theorem 3.10 for relation \( \vdash_{r} \). This fact allows us
to use the results of Section 3.4, in a simplified form, as tools for proving total cor-
rectness.
Theorem 3.17.

\[
\begin{align*}
\vdash_{I} & \{ \text{Pre} \} P \{ \text{Post} \} \\
\vdash_{I} & \{ \text{Pre} \} P \{ M_{P}^{L} \cap \text{Pre} \}.
\end{align*}
\]

The proof is analogous to that of Theorem 3.10. An immediate consequence of Theorem 3.7 is that \( \vdash_{I} \{ \text{Pre} \} P \{ \text{Post} \} \) implies the total correctness of \( P \) w.r.t. the specification \( (\text{Pre}, \text{sp}(P, \text{Pre})) \).

Theorem 3.18 (Total correctness). If \( \vdash_{I} \{ \text{Pre} \} P \{ \text{Post} \} \) holds, then \( P \) is totally correct w.r.t. the specification \( (\text{Pre}, \text{sp}(P, \text{Pre})) \).

Proof. By Theorem 3.17, \( \vdash_{I} \{ \text{Pre} \} P \{ \text{sp}(P, \text{Pre}) \} \) holds, since \( \text{sp}(P, \text{Pre}) = M_{P}^{L} \cap \text{Pre} \). From this and Theorem 3.16, we have that \( P \) is weak totally correct w.r.t. \( (\text{Pre}, \text{sp}(P, \text{Pre})) \). Again by the relation \( \text{sp}(P, \text{Pre}) = M_{P}^{L} \cap \text{Pre} \), we conclude that \( P \) is totally correct w.r.t. \( (\text{Pre}, \text{sp}(P, \text{Pre})) \). \( \square \)

In addition to total correctness, we show a form of completeness of the termination proof method associated with relation \( \vdash_{I} \). First, we adapt the notion of terminating programs used in the literature (see e.g. Ref. [13]), by taking into account preconditions.

Definition 3.9. For a program \( P \) and a set \( \text{Pre} \subseteq b_{L} \), we say that \( P \) is Pre-terminating iff for every \( A \in \text{Pre} \) the LD-tree for \( P \) and \( A \) is finite.

From the next Theorem, we have that, if \( \vdash_{I} \{ \text{Pre} \} P \{ \text{Post} \} \) holds then \( P \) is Pre-terminating iff \( \vdash_{I} \{ \text{Pre} \} P \{ \text{sp}(P, \text{Pre}) \} \) holds. In other words, the relation \( \vdash_{I} \) is a complete termination proof method, restricted to the triples in \( \vdash \).

Theorem 3.19 (Termination completeness). If \( \vdash_{I} \{ \text{Pre} \} P \{ \text{Post} \} \) holds then \( P \) is Pre-terminating. Conversely, \( \vdash_{I} \{ \text{Pre} \} P \{ \text{Post} \} \) and \( P \) Pre-terminating imply \( \vdash_{I} \{ \text{Pre} \} P \{ \text{sp}(P, \text{Pre}) \} \).

Proof. The if-part follows from Theorem 3.17, whereas the onlyif-part is reported in Appendix A as Theorem A.1 (i). \( \square \)

3.7.1. Well-supported interpretations

The notion of well-supported interpretation becomes simpler when considering relation \( \vdash_{I} \), in the particular case that \( \text{Post} \subseteq \text{Pre} \), it boils down to the well-known notion of supported interpretation (see e.g. Ref. [7]). As the next Theorem shows, the proof obligations on the level mapping used to prove \( \vdash_{I} \{ \text{Pre} \} P \{ \text{Post} \} \) implicitly satisfy part of the requirements of Definition 3.7. As a consequence, we obtain a simpler proof method.

Theorem 3.20. Assume that \( \vdash_{I} \{ \text{Pre} \} P \{ \text{Post} \} \) holds. Then the following statements are equivalent:

(i) \( \text{Post} \cap \text{Pre} = \text{sp}(P, \text{Pre}) \),
(ii) \( T_\Phi(\text{Post}) \supseteq \text{Post} \cap \text{Pre} \).
(iii) \( \text{Post} \) is well-supported w.r.t. \( P \) and \( \text{Pre} \).

Proof. (i) \( \rightarrow \) (ii) and (iii) \( \rightarrow \) (i) follow from Theorem 3.12. Let us prove (ii) \( \rightarrow \) (iii). We show that \( \text{Post} \) is a well-supported interpretation by considering the level mapping \( | \) used to prove \( \vdash_\phi \{\text{Pre}\} P \{\text{Post}\} \). Consider \( A \in \text{Post} \cap \text{Pre} \). As \( T_\Phi(\text{Post}) \supseteq \text{Post} \cap \text{Pre} \), there exists a clause \( A \leftarrow B_1, \ldots, B_n \in \text{ground}_\phi(P) \) such that \( \text{Post} \models B_1, \ldots, B_n \). Moreover, since \( \vdash_\phi \{\text{Pre}\} P \{\text{Post}\} \), we have that for every \( i \in [1..n] \), \( |A| > |B_i| \). Therefore \( \text{Post} \) is a well-supported interpretation w.r.t. \( P \) and \( \text{Pre} \). \( \square \)

Proof outlines for well-supportedness in the case of total correctness are simpler than those in the case of partial correctness. In practice, proof outlines are now obtained from Definition 3.8 simply by not considering the expressions \( i \), for \( i \in [0..n] \) and the related proof obligations.

3.7.2. Example: preorder tree traversal

As an example, we report the proof outlines for \text{PREORDER}, omitting those for \text{APPEND}.

\[
\{ \text{true} \}
\]
\[
\text{preorder} (\text{nil}, []).
\]

\[
\{ \text{true} \}
\]
\[
\text{preorder} (\text{leaf}(X), [X]).
\]

\[
\{ t(X, \text{Left}, \text{Right}) \in \text{Tree}(x, \beta) \land
Ls = [X| As] * Bs \text{ is a preorder traversal of it} \land |As| = ||Left|| \}
\]
\[
\text{preorder} (t(X, \text{Left}, \text{Right}), Ls) \leftarrow
\]
\[
\text{preorder} (\text{Left}, As).
\]
\[
\{ \text{Left} \in \text{Tree}(x, \beta) \land As \text{ is a preorder traversal of it} \}
\]
\[
\text{preorder} (\text{Right}, Bs).
\]
\[
\{ \text{Right} \in \text{Tree}(x, \beta) \land Bs \text{ is a preorder traversal of it} \}
\]
\[
\text{append} ([X| As], Bs, Ls).
\]
\[
\{ [X| As], Bs \in \text{GList} \land Ls = [X| As] * Bs \}
\]

The proof obligations of the last proof outline are fulfilled by noting that the preorder traversal of \( \text{Left} \) coincides with the sublist of \( Ls \) from position \( ||Left|| + 1 \), i.e. with \( As \), and similarly for \( \text{Right} \) and \( Bs \). We now check that every preorder-atom in \( \text{Post}_{\text{PREORDER}} \cap \text{Pre}_{\text{PREORDER}} \) is an instance of the head of some clause for which the assertion associated with the clause holds. Assume that \( \text{preorder}(t, Ls) \) is in \( \text{Post}_{\text{PREORDER}} \cap \text{Pre}_{\text{PREORDER}} \). Then:
- either \( t \) is \( \text{nil} \) and \( Ls \) is \( [ ] \). This case is covered by the first proof outline;
- or \( t \) is \( \text{leaf}(x) \) for some \( x \) and \( Ls \) is \( [x] \). This case is covered by the second proof outline;
- or \( t \) is \( \text{tree}(x, \text{left}, \text{right}) \) and \( Ls \) is \( [x] * bs \) where \( As \) is a preorder traversal of \( \text{left} \) and \( Bs \) is a preorder traversal of \( \text{right} \). In particular, \( As \) do coincide with \( ||\text{left}|| \), and then this case is covered by the third proof outline.

In conclusion, \text{PREORDER} is totally correct w.r.t. the specification

\[
(\text{Pre}_{\text{PREORDER}}, \text{Post}_{\text{PREORDER}} \cap \text{Pre}_{\text{PREORDER}})
\]
and since $\text{Post}_{\text{PREORDER}} \cap \text{Pre}_{\text{PREORDER}} = \text{Post}_{\text{PREORDER}}$, w.r.t. the specification 

$$(\text{Pre}_{\text{PREORDER}}, \text{Post}_{\text{PREORDER}}).$$

### 3.8. Weakest (liberal) preconditions

In contrast to the case of the strongest postconditions, we were unable to provide a simple direct characterization of the weakest (liberal) preconditions. However, in [49] we present a theory of the weakest (liberal) preconditions for logic programming, with the aim of describing $\text{wp}(P, \text{Post})$ and $\text{wp}(P, \text{Post})$ as fixpoints of the following operator $\text{wp}_{P, \text{Post}}$ which is defined on the lattice of Herbrand interpretations:

$$\text{wp}_{P, \text{Post}}(I) = \{ A \in B_L \mid \forall A \leftarrow B_1, \ldots, B_n \in \text{ground}(P):$$

$$\forall i \in [1, n]: \text{Post} \models B_1, \ldots, B_{i-1} \Rightarrow I \models B_i$$

$$\land \text{Post} \models B_1, \ldots, B_n \Rightarrow \text{Post} \models A \}.$$.

The main interest in $\text{wp}_{P, \text{Post}}$ is due to the following characterization of the weakest (liberal) precondition as the upward (resp., downward) ordinal closure of $\text{wp}_{P, \text{Post}}$.

We refer the reader to Ref. [49] for a proof of the Theorem, and additional properties of $\text{wp}_{P, \text{Post}}$.

**Theorem 3.21.** For a program $P$ and $\text{Post} \subseteq B_L$, the function $\text{wp}_{P, \text{Post}}$ is monotonic and downward continuous over the lattice $(2^B_L, \subseteq)$. Moreover:

(i) $\vdash \{ \text{Pre} \} P \{ \text{Post} \}$ iff $\text{Pre} \subseteq \text{wp}_{P, \text{Post}}(\text{Pre}).$

(ii) $\text{wp}(P, \text{Post}) = \text{gfp}(\text{wp}_{P, \text{Post}}) = \text{wp}_{P, \text{Post}} \uparrow \omega$.

(iii) $\text{wp}(P, \text{Post}) = \text{wp}_{P, \text{Post}} \uparrow \omega$.

### 4. Applications

In this section, we show how the proof method is of practical use for real applications. First, by appropriately defining $\text{Pre}$ and $\text{Post}$ for arithmetic built-in's, the proof method can be used to show the absence of run-time errors due to the selection of ill-typed arithmetic atoms. Second, we introduce a relation $\text{semi}$- $\vdash$, which, although equivalent to $\vdash$, is more suitable for proving correctness in a modular way. Also, we provide a sufficient condition, in the form of restrictions to the admissible $\text{Pre}$ and $\text{Post}$, which ensures the safe omission of the occur-check in the unification algorithm. Finally, we briefly overview some applications of the proof method to the verification of meta-programs, and some results on semantics decidability.

### 4.1. Arithmetic built-in's

A program with arithmetic is a logic program in which the predicates:

$$<, <=, =, =/, =, \text{is}, >, =$$

can appear only in clause bodies. These predicates are defined for particular terms, called ground arithmetic expressions (in short, gae's.) Formally, the set $\text{Gae}$ of gae's is obtained by removing from the Herbrand universe on the signature

$$\Sigma_{Ar} = \{ 0, 1, 2, \ldots, +2, \ast, /, \mod \}.$$
all the terms containing a division by zero. We denote by $Gae$ the set of numerals \{0, 1, -1, 2, -2 \ldots\}. For a gae $n$, $value(n)$ is the integer denoted by $n$. According to Ref. [54], we extend LD-resolution assuming that a program with arithmetic in which $>$ appear, implicitly contains the set of unit clauses whose heads are:

$$Post_> = \{ n > m \mid n, m \in Gae \land value(n) > value(m) \}$$

(and analogously for $<, \leq, \geq, \neq, \equiv$.) If is appears in the program, we consider the set:

$$Post_{is} = \{ value(m) \leftrightarrow m \mid m \in Gae \}.$$

Consider now a LD-derivation for a program with arithmetic and a query such that the atom $n > m$ is selected. If $n, m$ are gae's then, according to the implicit clauses, the LD-derivation fails if $value(n)$ is lower or equal than $value(m)$. If the value of $n$ is greater than that of $m$, the resolvent is the rest of the goal.

According to Ref. [54], we stipulate that a LD-derivation for a program with arithmetic and a query ends in an error if an atom $n > m$ is selected and $n, m$ are not gae's. This is the procedural semantics of $>$ in Prolog. A similar operational semantics is given for $\leq, \neq, \equiv, \equiv$, whereas for is only the second argument is required to be a gae.

Unfortunately, as discussed by Apt [8], it is not possible to reason in a declarative way on run-time arithmetic errors within the logic programming theory. In particular, the Lifting Lemma does not hold for programs with arithmetic. Consider now the program $PART$

\[
\begin{align*}
\text{part}(X, [Y|Xs], [Y|Ls], Bs) & \leftarrow X > Y, \text{part}(X, Xs, Ls, Bs). \\
\text{part}(X, [Y|Xs], [Y|Ls], [\sim|Bs]) & \leftarrow X \leq Y, \text{part}(X, Xs, Ls, Bs). \\
\text{part}(X, [], [], []) & .
\end{align*}
\]

for partitioning a list of gae's, and suppose that $\models \{Pre\} PART \{Post\}$ holds for some $Pre, Post$. By Corollary 3.1 (i), for every selected atom $A$, we have that $Pre \models A$. This suggests us a simple condition to prevent the selection of ill-typed arithmetic atoms, consisting of imposing that if $Pre \models A$ holds for an arithmetic atom $A$ then $A$ is correctly typed. For instance, if $n > m$ is selected and $Pre$ coincides on $>$-atoms with the set

$$Pre_> = \{ n > m \mid n, m \in Gae \}.$$

then $Pre \models n > m$ implies that $n, m$ are gae's, under the weak hypothesis that there exists at least one symbol $t$ in $\Sigma_\ell$ that does not belong to $\Sigma_{is}$. In fact, we notice that $n > m$ is ground, otherwise by instantiating the variables of $n, m$ with a ground term containing $t$ we get two terms that are not gae's. Since $ntt > m$ is ground, we have that $n > m \in Pre$ and then $n, m$ are gae's.

We reason analogously for the other arithmetic atoms, except for is, for which:

$$Pre_{is} = \{ t \leftrightarrow m \mid m \in Gae \}.$$

Considering again $PART$, we define the pre- and postcondition as follows:
\[ \text{Pre}_{\text{PART}} = \text{Pre}_> \cup \text{Pre}_\leq \cup \{ \text{part}(x, xs, Is, bs) \mid x \in \text{Gae} \land xs \in \text{List(Gae)} \} \]
\[ \text{Post}_{\text{PART}} = \text{Post}_> \cup \text{Post}_\leq \cup \{ \text{part}(x, xs, Is, bs) \mid x \in \text{Gae} \land xs, Is, bs \in \text{List(Gae)} \land Is < x \geq bs \} \].

By \( Is < x \geq bs \) we mean that every element in the list \( Is \) (resp., \( bs \)) is smaller (resp., greater or equal) than \( x \). It is readily checked that \( \vdash \{ \text{Pre}_{\text{PART}} \} \ \text{PART} \ \{ \text{Post}_{\text{PART}} \} \).

Therefore, when an arithmetic atom \( n > m \) is selected in a LD-derivation for \( \text{PART} \) and a query \( Q \) such that \( \vdash \{ \text{Pre}_{\text{PART}} \} \ Q \ \{ \text{Post}_{\text{PART}} \} \), we have that
\[ \text{Pre}_> \models n > m. \quad (2) \]

As discussed, this implies that \( n, m \) are gae’s and a fortiori that the LD-derivation does not end in an error. We generalize this reasoning by means of the following definition.

**Definition 4.1.** Let \( P \) be a program with arithmetic, and \( L \) such that \( \Sigma_L \setminus \Sigma_P \neq \emptyset \). We write \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) iff \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) holds for \( P \) as a logic program, and for every arithmetic predicate \( op \) appearing in \( P \), the sets of \( op \)-atoms in \( \text{Pre} \) and in \( \text{Post} \) coincide with \( \text{Pre}_{op} \) and \( \text{Post}_{op} \), respectively.

Under the hypothesis of Definition 3.2, we can show absence of run-time errors.

**Lemma 4.1.** Assume that \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) and \( \vdash \{ \text{Pre} \} \ Q \ \{ \text{Post} \} \) hold for a program with arithmetic \( P \) and a query \( Q \). Then no LD-derivation for \( P \) and \( Q \) ends in an error.

Since LD-trees of programs with arithmetic are still finitely branching, by the Lemma above we can extend the lifting lemma and the strong completeness theorem for LD-resolution to programs with arithmetic \( P \) such that \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) holds. As a consequence, the proof theory of Section 3 and the related results can be generalized to programs with arithmetic.

### 4.2. Modular verification

The definition of relation \( \vdash \), has a major drawback, due to the lack of expressiveness of level mappings in modular correctness proofs. We introduce the problem with an example. Consider the program \textsc{SubList}.

\[
\text{sublist}(Xs, Ys) \leftarrow Xs \text{ is a sublist of } Ys.
\]
\[
\text{sublist}(Xs, Ys) \leftarrow \\
\text{append}(\_, Zs, Ys), \\
\text{append}(Xs, \_, Zs).
\]

augmented by the \textsc{Append} program.

A level mapping such that
\[
|\text{append}(xs, xs, zs)| = |zs|
\]
is a natural candidate to show \( \vdash \{ \text{Pre}^\prime_{\text{APPEND}} \} \ \text{APPEND} \ \{ \text{Post}^\prime_{\text{APPEND}} \} \), where
\[
\text{Pre}^\prime_{\text{APPEND}} = \{ \text{append}(xs, xs, zs) \mid zs \in \text{GList} \}.
\]
Analogously, when considering the specification for SUBLIST:

\[ \text{Pre}_{\text{SUBLIST}} = \{ \text{sublist}(\mathbf{x}_s, \mathbf{y}_s) \mid \mathbf{x}_s \in \mathbf{GList} \} \cup \text{Pre}_{\text{APPEND}} \]
\[ \text{Post}_{\text{SUBLIST}} = \{ \text{sublist}(\mathbf{x}_s, \mathbf{y}_s) \mid \text{sublist of } \mathbf{y}_s \text{ in } \mathbf{GList} \} \cup \text{Post}_{\text{APPEND}} \]

an intuitively correct level mapping is such that \(|\text{sublist}(\mathbf{x}_s, \mathbf{y}_s)| = |\mathbf{y}_s|\). However, the proof obligations of Definition 3.2 require that

\[ |\text{sublist}(\mathbf{x}_s, \mathbf{y}_s)| > |\text{append}(\mathbf{x}_s, \mathbf{z}_s, \mathbf{y}_s)| = |\mathbf{y}_s|. \]

Therefore, to show \( \vdash \{ \text{Pre}_{\text{SUBLIST}} \} \text{ SUBLIST } \{ \text{Post}_{\text{SUBLIST}} \} \), we must consider a somewhat unnatural level mapping such as \(|\text{sublist}(\mathbf{x}_s, \mathbf{y}_s)| = |\mathbf{y}_s| + 1\). Unfortunately, such a phenomena propagates upward when considering programs which use SUBLIST, giving rise to counter-intuitive level mappings and preventing modular program development.

A relation \( \text{semi-} \vdash \), is introduced in Ref. [46] following the approach of Ref. [5], which addresses the modularity problems shown above.

First, for two predicate symbols \( p \) and \( q \), we write \( p \vdash q \) if \( p \) uses \( q \) in its definition, but \( p \) and \( q \) are not mutually recursive; we write \( p \vdash_\| q \) if \( p \) and \( q \) are mutually recursive. \( \text{rel}(A) \) denotes the predicate symbol of the atom \( A \).

**Definition 4.2.** Given a program \( P \) and a specification \( (\text{Pre}, \text{Post}) \), we write \( \text{semi-} \vdash \{ \text{Pre} ; P \} \{ \text{Post} \} \) iff there exists a level mapping \( | \mid \) such that for every 
\[
A \leftarrow B_1, \ldots, B_n \in \text{ground}_L(P)
\]
1. for \( i \in [1, n] \):
   \[
   \text{Pre} \models A \land \text{Post} \models B_1, \ldots, B_{i-1} \Rightarrow
   \begin{align*}
   (a) & \text{Pre} \models B_i \text{ and } \\
   (b) & |A| > |B_i| \text{ if } \text{rel}(A) \equiv \text{rel}(B) \\
   & |A| \geq |B_i| \text{ if } \text{rel}(A) \preceq \text{rel}(B)
   \end{align*}
\]
2. \( \text{Pre} \models A \land \text{Post} \models B_1, \ldots, B_n \Rightarrow \text{Post} \models A. \)

In contrast to Definition 3.2, in 1(b) we now distinguish two cases depending whether \( \text{rel}(A) \) and \( \text{rel}(B_i) \) are (or not) mutually recursive predicates. If they are mutually recursive, a strict decreasing is imposed.

Consider the SUBLIST program again. By defining

\[
|\text{append}(\mathbf{x}_s, \mathbf{y}_s, \mathbf{z}_s)| = |\mathbf{z}_s| \\
|\text{sublist}(\mathbf{x}_s, \mathbf{y}_s)| = |\mathbf{y}_s|
\]

we have that \( \text{semi-} \vdash \{ \text{Pre}_{\text{SUBLIST}} \} \text{ SUBLIST } \{ \text{Post}_{\text{SUBLIST}} \} \) holds. In fact, since sublist and append are not mutually recursive, the decreasing of the level mapping has not to be strict.

In [46], it is shown that relations \( \vdash \), and \( \text{semi-} \vdash \), coincide, in the following sense:

\( \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds iff \( \text{semi-} \vdash \{ \text{Pre} \} P \{ \text{Post} \} \) holds.

This result allows us to extend all of the properties and tools (such as proof outlines) of triples in relation \( \vdash \), to triples in \( \text{semi-} \vdash \). The wide applicability of relation \( \text{semi-} \vdash \), is supported by several results on modular program verification. We refer the reader to [46] for methods to prove that a triple \( \{ \text{Pre} \} P \cup P' \{ \text{Post} \} \) is in a relation \( \vdash \), or \( \text{semi-} \vdash \), starting from proofs that triples for \( P \) and \( P' \) are in the same relation.
4.3. The occur-check problem

Apt and Pellegrini Ref. [6] present a methodology for the safe omission of the occur-check in the Martelli-Montanari unification algorithm. Most of Prolog interpreters omit the occur-check for efficiency reasons: unfortunately, this means that the correctness of LD-resolution is lost. However, Apt and Pellegrini show that for many practical programs, the occur-check omission is safe, by providing some sufficient conditions. First, we introduce some basic definitions.

Definition 4.3.

- Consider a n-ary predicate symbol \( p \). A mode for \( p \) is a function \( d_p \) from \( \{1, \ldots, n\} \) to \( \{+, -\} \). If \( d_p(i) = '+' \) we call \( i \) an input position. If \( d_p(i) = '-' \) then \( i \) is called an output position (with respect to \( d_p \)). We write \( d_p \) in the more suggestive form \( p(d_p(1), \ldots, d_p(n)) \).

- An atom is called output-linear if the family of terms which occur in its output positions is linear, i.e. no variable occurs twice in the family.

- A pair of atoms \((A, B)\) is NSTO (not subject to occur-check) if in every computation of the Martelli-Montanari algorithm the occur-check yields false.

A sufficient condition of Ref. [6] can be integrated within the proof theory based on relation \( \triangleright \) in order to show the safe omission of the occur-check along a LD-derivation.

Theorem 4.1. Assume that \( \triangleright \{\text{Pre}\} \cap \text{Post}\) and \( \triangleright \{\text{Pre}\} \supset \text{Post}\) hold. Consider a set \( \Pi \subseteq \Pi_L \) of predicate symbols such that

(i) for every atom \( A \) such that \( \text{rel}(A) \in \Pi \), if \( \text{Pre} \models A \) then only ground terms appear in the input positions of \( A \) and

(ii) the head of every clause from \( P \) whose predicate symbol is in \( \Pi \) is output linear.

Then for every atom \( A \) such that \( \text{rel}(A) \in \Pi \) and selected in a LD-derivation for \( P \) and \( Q \), the omission of occur-check in the Martelli-Montanari unification algorithm is safe.

Proof. See Ref. [47], Theorem 4.9. \( \square \)

4.4. Meta-interpreters

Meta-circular interpreters have been introduced as a fundamental feature of advanced programming languages. Since the early studies, many meta-interpreters have been proposed and proved correct with respect to their intended behavior. However, the task of proving correctness has been largely performed using ad-hoc techniques, depending case by case on the semantics, the particular meta-program and the range of properties one was interested in verifying.

In Ref. [48], a general criterion is introduced for reasoning about meta-interpreters. The basic idea is to apply the general purpose verification methods based on relations \( \triangleright \) and \( \triangleright \), to the case study of the Vanilla meta-interpreter, and, more generally, to generic meta-interpreters, by relating the pre- and postconditions of the object program to those of the meta-program. The main results of Ref. [48] can be summarized as follows: under certain natural assumptions, all in-
interesting verification properties lift up from the object program to Vanilla, including
• (weak) partial correctness,
• (weak) total correctness,
• absence of arithmetic errors,
• call pattern characterization,
• correct and computed instances characterization.

4.5. Semantics decidability

The semantics decidability issue has been largely investigated in the literature with respect to the $\mu$-semantics [15]. In our context, a by-result of Theorem 3.15 is that, when $\vdash \{\text{Pre}\} P \{\text{Post}\}$ hold then it is decidable for every $A \in \text{Pre}$ whether $A \in \mu(P)$.

Recently, the decidability of the $\\ell$ and $\mathcal{Y}$-semantics has been investigated in Ref. [50]. A Prolog implementation of a decision procedure is presented for the class of acceptable logic programs, namely programs $P$ such that $\vdash \{B_k\} P \{\text{Post}\}$ holds for some $\text{Post}$. Moreover, semantics decidability and program testing are shown to be strongly related, and, in practice, the proposed decision procedure is the core of a test driver. We generalize those decidability results to programs such that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds.

**Theorem 4.2.** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds, and consider an atom $A$ such that $\text{Pre} \models \exists A$. Then

(i) it is decidable whether $A \in \ell(P)$;
(ii) it is decidable whether $A \in \mathcal{Y}(P)$.

**Proof.** See Ref. [47], Theorem 4.13. □

5. General logic programs

General programs and queries are introduced by allowing negated atoms in the body of clauses and queries. In this section, we extend the proof theory to reason on general programs and queries. Many results cannot be lifted directly, due to some well-known problems with extending the logic programming theory to handle negation. In particular, a major difficulty is the incompleteness of the negation as failure rule w.r.t. Clark’s completion semantics $\text{comp}(P)$ (see Ref. [41]). We partly solve this problem by reasoning on the basis of the $\vdash$ relation. This section is structured as follows: first, we consider negated atoms only in queries, dealing with the so-called LDNF-resolution. Then we extend the approach to general programs, by providing a method for (weak) total correctness. As an application, we also obtain a rather general completeness result for LDNF-resolution.

5.1. LDNF-resolution

We start by considering negation only in queries. Following Apt [7], we introduce LDNF-resolution.
Definition 5.1.
- A LDNF-derivation is a SLDNF-derivation for a program and a general query, by using the leftmost selection rule.
- We write $\vdash_{L} \{\text{Pre}\} Q \{\text{Post}\}$ for a general query $Q$ iff there exist a level mapping $| |$ and $k \in \mathbb{N}$ such that for every ground instance $L_1, \ldots, L_n$ of $Q$:
  for $i \in [1,n]$
  $$
  \text{Post} \models L_1, \ldots, L_{i-1} \Rightarrow \begin{cases} 
  \text{Pre} \models A_i \land k > |A_i| & \text{if } L_i = A_i, \\
  \text{Pre} \models A_i \land k > |A_i| & \text{if } L_i = \neg A_i.
  \end{cases}
  $$
- Given a program $P$ and a general query $Q$, we say $P \cup \{Q\}$ does not flounder if there is no LDNF-derivation for $P$ and $Q$ in which a non-ground negative literal is selected.

In the following theorem, the completeness of negation as failure rule is shown for (positive) programs $P$ and general queries $Q$ that are in the $\vdash_{L}$ relation.

Theorem 5.1. Let $P$ be a program and $Q$ a general query such that $\vdash_{L} \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash_{L} \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping. If:
- $P \cup \{Q\}$ does not flounder, and
- $\text{comp}(P) \models Q$ for $Q$'s instance of $Q$,

then there exists a LDNF-refutation for $P$ and $Q$ with computed instance more general than $Q$.

Proof. First, we point out that by Theorem 3.10 $\vdash_{L} \{\text{Pre}\} P \{M^P_{\text{F}} \cap \text{Pre}\}$ and $\vdash_{L} \{\text{Pre}\} Q \{M^Q_{\text{F}} \cap \text{Pre}\}$ hold. Let us prove that if $\text{comp}(P) \models Q'$ then there exists a LDNF-refutation for $P$ and $Q'$ with computed instance $Q'$. The proof is by induction on the number $n$ of literals in $Q'$.

(Base) If $Q'$ consists of one literal then we distinguish two cases. If $Q' = A$ then by the Strong Completeness Theorem of SLD-resolution there exists a LD(NF-) refutation for $P$ and $A$. If $Q' = \neg A$ then $A$ is ground since $P \cup \{Q'\}$ does not flounder – otherwise $P \cup \{Q\}$ flounders. Since $\vdash_{L} \{\text{Pre}\} A \{M^A_{\text{F}} \cap \text{Pre}\}$, by Theorem 3.15 the LD-tree for $P$ and $A$ is finite. Moreover, it is finitely failed. Otherwise, by correctness of LDNF-resolution $\text{comp}(P) \models A$: this is in contradiction with the assumption $\text{comp}(P) \models \neg A$ and the fact that $\text{comp}(P)$ is consistent. Thus there exists a LDNF-refutation for $P$ and $Q'$.

(Step) We distinguish two cases.
- If $Q' = A, Q''$ then by the Strong Completeness Theorem of SLD-resolution there exists a LD(NF-) refutation for $P$ and $A$. Moreover, by Corollary 3(ii) $M^P_{\text{F}} \cap \text{Pre} \models A$ and then $\vdash_{L} \{\text{Pre}\} Q'' \{M^P_{\text{F}} \cap \text{Pre}\}$. The result follows by applying the inductive hypothesis on $Q''$.
- If $Q' = \neg A, Q''$ then $A$ is ground since $P \cup \{Q'\}$ does not flounder – otherwise $P \cup \{Q\}$ flounders. Since $\vdash_{L} \{\text{Pre}\} A \{M^A_{\text{F}} \cap \text{Pre}\}$, the LD-tree for $P$ and $A$ is finite. Moreover, it is finitely failed, i.e. $A \subseteq FF^P_{\text{F}}$. Otherwise, by correctness of LDNF-resolution $\text{comp}(P) \models A$: this is in contradiction with the assumption $\text{comp}(P) \models \neg A, Q''$ and the fact that $\text{comp}(P)$ is consistent. Thus $Q''$ is the LDNF-resolvent of $Q$. Moreover, $A \subseteq FF^P_{\text{F}}$ implies $M^A_{\text{F}} \cap \text{Pre} \models \neg A$, which in turn implies $\vdash_{L} \{\text{Pre}\} Q'' \{M^A_{\text{F}} \cap \text{Pre}\}$. The conclusion follows by applying the inductive hypothesis on $Q''$. 
Since \( P \cup \{Q\} \) does not flounder, we can apply the (LDNF\(^{-}\)-version of the) Lifting Lemma to the refutation for \( P \) and \( Q' \), thus obtaining a LDNF\(^{-}\)-refutation for \( P \) and \( Q \) with computed instance more general than \( Q' \). \( \square \)

5.2. LDNF-resolution

5.2.1. Correctness

The exposition of the approach for general programs is organized as follows: first, we extend to general programs the definitions of (weak) total correctness, and of relation \( \vdash \). Second, we show some correctness properties, including persistency, termination and (weak) total correctness. Finally, we concentrate on completeness of SLDNF-resolution, by exploiting the correctness results. Therefore, we start by extending Definition 2.2.

**Definition 5.2.** Consider a general program \( P \).

(i) LDNF-resolution is SLDNF-resolution together with the *leftmost* selection rule.

(ii) We denote with \( M^l_p \) the set of \( A \in B_t \) such that there exists a LDNF-refutation for \( P \) and \( A \), and with \( FF^l_p \) the set of \( A \in B_t \) such that there exists a finitely failed LDNF-tree for \( P \) and \( A \). \( FF^l_p \) is the set \( B_t \setminus FF^l_p \).

(iii) Given a general program \( P \) and a general query \( Q \), we say \( P \cup \{Q\} \) does not *flounder* if there is no LDNF-derivation for \( P \) and \( Q \) where a non-ground negative literal is selected.

(iv) \( P \) is *weak totally correct* w.r.t. a specification \((Pre, Post)\) iff \( M^l_p \cap Pre \subseteq Post \) and \( Pre \subseteq M^l_p \cup FF^l_p \).

(v) \( P \) is *total correct* w.r.t. a specification \((Pre, Post)\) iff \( M^l_p = Post \) and \( Pre \subseteq M^l_p \cup FF^l_p \).

Although \( M^l_p \) and \( FF^l_p \) now are not declaratively defined, we will show later that for the class of programs we are interested in they have the expected declarative interpretation. Regarding queries, the definition of relation \( \vdash \) remains the same as Definition 5.1. Relation \( \vdash \) is extended to general programs as follows.

**Definition 5.3.** Let \( P \) be a general program, and \((Pre, Post)\) a specification.

We write \( \vdash_{\{Pre\}} P \{Post\} \) iff there exists a level mapping \( | \) such that:

(i) for every \( A \leftarrow L_1, \ldots, L_n \in \text{ground}_i(P) \):

1. for \( i \in [1..n] \):

\[ Pre \models A \land Post \models L_1, \ldots, L_i \]  

\[ \left\{ 
\begin{array}{l}
Pre \models B_i \land |A| > |B_i| & \text{if } L_i = B_i \\
Pre \models B_i \land |A| > |B_i| & \text{if } L_i = \neg B_i \\
\end{array}
\right.
\]

2. \( Pre \models A \land Post \models L_1, \ldots, L_n \Rightarrow Post \models A \)

(ii) \( T_p(\text{Post}) \supseteq Post \cap Pre \).

We say that \( P \) is *non-floundering* (w.r.t. \( Pre \)) iff for every \( A \in Pre, P \cup \{A\} \) does not flounder.

It is worth noting that relation \( \vdash \), for general programs is not a conservative extension of that for logic programs. In general, \( \vdash_{\{Pre\}} P \{Post\} \) for a logic program
$P$ does not necessarily imply that $\vdash, \{Pre\} P \{Post\}$ holds for $P$ as a general program. This is due to the additional requirement (ii) of Definition 5.3

$$T_P(\text{Post}) \supseteq \text{Post} \cap \text{Pre}.$$  

However, we point out that we already met a similar relation: by Theorem 3.20, it is equivalent for logic programs to require that $\text{Post}$ is well-supported w.r.t. $P$ and $\text{Pre}$, i.e. that $\text{Post} \cap \text{Pre} = M^*_P \cap \text{Pre}$. Therefore, by Theorem 3.17, if $\vdash, \{Pre\} P \{Post\}$ holds for a logic program then

$$\vdash, \{Pre\} P \{M^*_P \cap \text{Pre}\}$$

holds considering $P$ as a general program. In other words, when dealing with triples for general programs, we are forced to consider the strongest postconditions.

If condition (ii) of Definition 5.3 is omitted, then we are not able to show the basic properties of correctness, such as the equivalent of Corollary 3.1 (ii) for general programs. Consider, as an example the program $P$:

$$p \leftarrow \neg q.$$  

and the query $\neg q$. Condition (i) holds for $P$ and $\text{Pre} = \text{Post} = \{p, q\}$, and $\vdash, \{\text{Pre}\} \neg q \{\text{Post}\}$. However, $\text{Post} \not\models \neg q$ even though $\neg q$ is a computed instance of $P$ and $\neg q$.

In the following theorem, we extend the properties of persistency and call pattern characterization to general programs. As we will see in an example, call pattern characterization is essential for establishing non-floundering of general programs.

**Theorem 5.2 (Persistency and call patterns).** Assume that $\vdash, \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash, \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping $!$. Then:

(i) for every LDNF-resolvent $Q'$ of $P$ and $Q$, $\vdash, \{\text{Pre}\} Q' \{\text{Post}\}$ holds by the same $| |$;

(ii) for every literal $A$ or $\neg A$ selected in a LDNF-derivation for $P$ and $Q$, $\text{Pre} \models A$;

(iii) for every computed instance $Q'$ of $P$ and $Q$, $\text{Post} \models Q'$.

**Proof.** (i) See Lemma B.2 (iii) reported in Appendix B.

(ii) Immediate by (i) and Definition 5.1.

(iii) Let $x_1, \ldots, x_n$ be the variables of $Q$, and $p$ a fresh predicate symbol of arity $n$. We define $\text{Pre}'$ as $\text{Pre}\cup\{p(t_1, \ldots, t_n) \mid \text{Post} \models Q(x_i/t_i \mid i \in [1..n]) \}$ and $\text{Post}'$ as $\text{Post}\cup p(U_1 \times \cdots \times U_n)$. With this assumptions, it is readily checked that $\vdash, \{\text{Pre}'\} P\cup\{p(x_1, \ldots, x_n) \leftarrow Q \}$ $\{\text{Post}'\}$, and $\vdash, \{\text{Pre}'\} Q, p(x_1, \ldots, x_n) \{\text{Post}'\}$, by fixing the level of any $p(t_1, \ldots, t_n)$ to the natural number $k$ provided by Definition 5.1. By hypothesis, there exists a LDNF-derivation for $P$ and $Q, p(x_1, \ldots, x_n)$ where $p(T_1, \ldots, T_n)$ is selected, with $Q(x_i/T_i \mid i \in [1..n])$ variant of $Q'$. By (ii) and the definition of $\text{Pre}'$, we conclude that $\text{Post} \models Q(x_i/T_i \mid i \in [1..n])$, and then $\text{Post} \models Q'$. □

As a consequence, the Termination Theorem 3.15 extends to general programs.

**Theorem 5.3 (Termination).** Assume that $\vdash, \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash, \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping. Then the LDNF-tree for $P$ and $Q$ is finite.
Proof. The proof is by induction on the rank of the LDNF-tree. If the rank is 0, then by Lemma B.2 (ii) there cannot be an infinite branch. Since the LDNF-tree is finitely branching, by König's Lemma, it is finite. If the rank is greater than 0, then by inductive hypothesis the rank of a subsidiary tree used in a LDNF-derivation is lower and then, by inductive hypothesis, finite. Moreover, by Lemma B.2 (ii) there cannot be an infinite branch. Since the LDNF-tree is finitely branching, by König's Lemma, it is finite.

However, stating that every LDNF-tree of $P$ and $Q$ is finite does not necessarily mean that the Prolog computation for $P$ and $Q$ eventually terminates. For the program

\[ \text{NIL} \]

\[ p \leftarrow \neg p. \]

and the query $p$, we have that there exists no LDNF-tree. Therefore, every LDNF-tree is finite. In contrast, the Prolog computation runs forever, by trying to built a subsidiary tree for $p$, each time $\neg p$ is selected. Another difference between LDNF-resolution and Prolog is that the latter does not check for floundering. We refer the reader to the paper of Apt and Doets [3] for a discussion on the differences between LDNF-resolution and Prolog.

Here, we confine ourselves to observe that with the assumptions of Theorem 5.3, termination w.r.t. the Prolog computation can be shown. This follows by observing that each time a subsidiary tree for $\neg A$ is being computed, its root is $A$, i.e. it is a positive literal. Therefore, there is at least one child, due to the resolution of $A$ with some clause. Therefore, if there is an infinite sequence $\{A_i\}_{i \geq 0}$ of atoms such that $A_i$ is the root of a subsidiary tree being computed during the resolution of $A$, then, by Lemma B.2 (ii), $|A_{i+1}| \preceq_m |A_i|$, where $| |$ maps queries into bags of natural numbers. We conclude that the sequence above cannot be infinite, since bags of natural numbers is a well-founded order.

In particular, $\vdash_\vartriangleright \{\text{Pre}\} \ \text{NIL} \ \{\text{Post}\}$ does not hold for $(\text{Pre}, \text{Post})$ such that $p \in \text{Pre}$, since we would have to show $|p| > |p|$. Some relevant properties of relation $\vdash_\vartriangleright$ are summarized in the following lemma.

Lemma 5.1. Assume that $\vdash_\vartriangleright \{\text{Pre}\} \ A \ \{\text{Post}\}$ holds and $P$ is non-floundering. The following statements hold:

(i) $\text{Pre} \subseteq M^L_P \cup FF^L_P$,
(ii) $M^L_P \cap \text{Pre} = FF^L_P \cap \text{Pre}$,
(iii) $M^L_P \cap \text{Pre} \subseteq \text{Post}$,
(iv) $FF^L_P \cap \text{Pre} \subseteq \text{Post}^*$,
(v) $\vdash_\vartriangleright \{\text{Pre}\} \ A \ \{\text{Post} \cap \text{Pre}\}$.

Proof. (i) For every $A \in \text{Pre}$, by Theorem 5.3 the LDNF-tree for $P$ and $A$ is finite. Since $P$ is non-floundering, then either there exists a LDNF-refutation or the LDNF-tree is finitely failed, i.e. $A \in M^L_P \cup FF^L_P$. Therefore, $\text{Pre} \subseteq M^L_P \cup FF^L_P$.

(ii) The $\subseteq$ inclusion holds since $M^L_P \subseteq FF^L_P$ by definition of $M^L_P$ and $FF^L_P$. On the other hand, by (i), $FF^L_P \cap \text{Pre}$ is included in $M^L_P$. Therefore $M^L_P \cap \text{Pre} = FF^L_P \cap \text{Pre}$.

(iii) Consider $A \in M^L_P \cap \text{Pre}$. By definition of $M^L_P$ there exists a LDNF-refutation for $P$ and $A$. By Lemma B.1(ii), $A \in \text{Post}$. Thus, $M^L_P \cap \text{Pre} \subseteq \text{Post}$. 

(iv) Consider \( A \in FF_p^L \cap Pre \). By definition of \( FF_p^L \) there exists a LDNF-refutation for \( P \) and \( \neg A \). By Lemma B.1 (ii), \( Post \models \neg A \). Therefore, \( FF_p^L \cap Pre \subseteq Post' \).

(v) Definition 5.3 (i) holds by reasoning in the same way of Theorem 3.7. Let us verify Definition 5.3 (ii). We have to show that \( T_P (Post \cap Pre) \supseteq Post \cap Pre \). Consider \( A \in Post \cap Pre \). Since \( \vdash_{ } \{ Pre \} \models P \{ Post \} \) holds, there exists
\[
A \leftarrow L_1, \ldots, L_n \in \text{ground}_L(P).
\]
such that \( Post \models L_1, \ldots, L_n \). We show that for \( i \in [1,n] \) \( Post \cap Pre \models L_i \). Consider two cases.
- If \( L_i \) is a positive literal, say \( B \), then by Definition 5.3 (i), \( Pre \models B \). Therefore \( Post \cap Pre \models L_i \).
- If \( L_i \) is a negative literal then \( Post \models L_i \) implies \( Post \cap Pre \models L_i \).

The first consequence of the lemma is the weak total correctness theorem for general programs.

**Theorem 5.4 (Weak total correctness).** If \( \vdash_{ } \{ Pre \} \models P \{ Post \} \) holds and \( P \) is non-floundering then \( P \) is weak totally correct w.r.t. the specification \( \{ Pre, Post \} \).

**Proof.** The conclusion is an immediate consequence of Lemma 5.1 (i, iii).

Similarly to positive programs, we are in the position to define the notion of strongest postcondition for \( P \) and \( Pre \) as the intersection of all postconditions \( Post \) such that \( \vdash_{ } \{ Pre \} \models P \{ Post \} \). We still denote it by \( sp(P, Pre) \). However, in the case of general programs, the notion of strongest postcondition does not result in a relevant concept. As showed in Theorem 3.20, the inclusion \( T_P (Post) \supseteq Post \cap Pre \) required in Definition 5.3 is equivalent for positive programs to force \( Post \cap Pre = M_p \cap Pre = sp(P, Pre) \). This fact extends to general programs, as pointed out by the following theorem.

**Theorem 5.5.** Assume that \( \vdash_{ } \{ Pre \} \models P \{ Post \} \) holds and \( P \) is non-floundering. Then:

(i) \( M_P^L \cap Pre = Post \cap Pre \),

(ii) \( FF_P^L \cap Pre = Post' \cap Pre \),

(iii) \( \vdash_{ } \{ Pre \} \models P \{ M_P^L \cap Pre \} \).

**Proof.** (i) The \( \subseteq \) inclusion is shown in Lemma 5.1 (iii). Moreover:

\[
Post \cap Pre
\]
\[
\subseteq Post \cap ( (M_P^L \cap Pre) \cup (FF_P^L \cap Pre) )
\]
\[
\text{Distributivity}
\]
\[
= (Post \cap M_P^L \cap Pre) \cup (Post \cap FF_P^L \cap Pre)
\]
\[
\{ \text{Lemma 5.1 (iii), (iv)} \}
\]
\[
= M_P^L \cap Pre.
\]

(ii) This is a direct consequence of (i) and of Lemma 5.1 (ii).

(iii) By Lemma 5.1 (v), \( \vdash_{ } \{ Pre \} \models P \{ Post \cap Pre \} \) holds. By (i), \( Post \cap Pre \) coincides with \( M_P^L \cap Pre \). Therefore, \( \vdash_{ } \{ Pre \} \models P \{ M_P^L \cap Pre \} \) holds.
We are now in the position to state the total correctness theorem for general programs.

**Theorem 5.6 (Total correctness).** If $\vdash \{\text{Pre}\} \ P \ \{\text{Post}\}$ holds and $P$ is non-floundering, then $P$ is totally correct w.r.t. the specification $(\text{Pre}, \text{Post} \cap \text{Pre})$.

**Proof.** By Lemma 5.5, (i, iii), $\vdash \{\text{Pre}\} \ P \ \{\text{g#nere}\} \ P \ \{\text{Post} \cap \text{Pre}\}$ holds, with $M_P \cap \text{Pre} = \text{Post} \cap \text{Pre}$. By Theorem 5.4, $P$ is weak totally correct w.r.t. the specification $(\text{Pre}, \text{Post} \cap \text{Pre})$. Noting that $M_P \cap \text{Pre} = \text{Post} \cap \text{Pre}$, by Definition 5.2, we conclude that $P$ is totally correct w.r.t. the specification $(\text{Pre}, \text{Post} \cap \text{Pre})$. 

We conclude by pointing out that notions and results such as proof outlines, reasoning on arithmetic built-in's, and modular proofs, directly extend to general programs.

5.2.2. Example: transitive closure

Consider the following program TRANS, used to calculate the transitive closure of a given relation.

\[
\text{trans}(x, y, e, v) \leftarrow x \rightarrow_{e \cap \varphi} y \\
\text{trans}(X, Y, E, V) \leftarrow \\
\quad \text{member}([X, Y], E), \\
\quad \neg \text{member}'(X, V). \\
\text{trans}(X, Z, E, V) \leftarrow \\
\quad \text{member}([X, Y], E), \\
\quad \neg \text{member}'(X, V), \\
\quad \text{trans}(Y, Z, E, [X\mid V]). \\
\text{member}(X, [X\mid T]). \\
\text{member}(X, [Y\mid T]) \leftarrow \\
\quad \text{member}(X, T). \\
\text{member}'(X, [X\mid T]). \\
\text{member}'(X, [Y\mid T]) \leftarrow \\
\quad \text{member}'(X, T).
\]

The definitions of \text{member} and \text{member}' coincide, and, in practice, they are not replicated. However, the uses (or \text{directionalities}) highlighted by that distinction will be useful to shown absence of floundering. Let $e$ be a binary relation on a set of constants $\alpha$, represented as an element in $\text{List}([x, z])$, i.e. a list of pairs $[x, y]$ with $x, y \in \alpha$. For $x \in \alpha$, the intended meaning of a query such as $\text{trans}(x, y, e, [ ])$ is to find out all $[x, y]$ in the transitive closure of $e$. To define suitable pre- and postconditions, we write $x \rightarrow_{e \cap \varphi} y$ when there is a path from $x$ to $y$ in $e$ that does not traverse pairs $[a, b]$ such that $a$ is in the list $v$. We define:

\[
\text{Pre} = \{ \text{trans}(x, y, e, v) \mid x \in \alpha \land v \text{ list of distinct elements in } \alpha \land \\
\quad v \in \text{List}([x, z]) \land \forall a \in v \exists [a, b] \text{ in } e \} \cup \\
\{ \text{member}(x, Is) \mid x \in [x, U], \ Is \in \text{List}([x, z]) \} \cup \\
\{ \text{member}'(x, v) \mid x \in \alpha, \ v \in \text{List}(\alpha) \}.
\]
\[ \text{Post} = \{ \text{trans}(x, y, e, v) \mid x \rightarrow^e v, y \} \cup \{ \text{member}(x, Is) \mid x \text{ is in } Is \in \text{GList} \} \cup \{ \text{member'}(x, Is) \mid x \text{ is in } Is \in \text{GList} \}. \]

Next, we define the following level mapping:

\[
|\text{member}(x, e)| = |\text{member'}(x, e)| = |e| \\
|\text{trans}(x, y, e, v)| = 2 \cdot |e| - |v| + 1
\]

for atoms in \( \text{Pre} \), and 0 elsewhere. We note that the level mapping is well-defined, since for \( \text{trans}(x, y, e, v) \in \text{Pre} \), we have that \(|e| \geq |v|\) and then

\[ |\text{trans}(x, y, e, v)| \geq 0. \]

We observe that in the case \( \text{Pre} = B_k \), we would have needed a more complicated level mapping. Such a case is shown in [13], where the focus was on termination. Here, our precondition simplifies the definition of the level mapping considerably, albeit large enough to reason on the interesting queries, such as \( \text{trans}(x, y, e, [\_]) \). Proving that \( \vdash \{\text{Pre}\} \text{ TRANS } \{\text{Post}\} \) holds is handy. As an example, we show a proof obligation relatively to the second clause. Consider a ground instance:

\[
\text{tran}(x, z, e, v) \\
\quad \quad \text{member}([x, y], e), \\
\quad \quad \neg \text{member'}(x, v), \\
\quad \quad \text{trans}(y, z, e, [x \mid v]).
\]

such that \( \text{Pre} \models \text{trans}(x, z, e, v) \) and

\[ \text{Post} \models \text{member}([x, y], e), \neg \text{member'}(x, v). \]

We have to show that:

\[ \text{Pre} \models \text{trans}(y, z, e, [x \mid v]) \quad \text{and} \quad (3) \\
|\text{trans}(x, z, e, v)| > |\text{trans}(y, z, e, [x \mid v])|. \]

For (3), we have that since \([x, y]\) is in \( e \) and \( e \) is a list of pairs of elements in \( z \), then \( y \) is in \( z \). In addition, \([x \mid v]\) is a list of distinct elements in \( z \), since \( v \) is a list of distinct elements and \( x \) is not in \( v \). Finally, for \( \forall a \in v \exists [a, b] \in e \) in \( e \) and \([x, y]\) is in \( e \) imply \( \forall a \in [x \mid v] \exists [a, b] \in e \). Therefore, (3) holds.

For (4), we note that \(|\text{trans}(x, z, e, v)| = 2 \cdot |e| - |v| + 1 > 2 \cdot |e| - (|v| + 1) + 1 = |\text{trans}(y, z, e, [x \mid v])|\).

Another useful observation in showing the proof obligations relatively to the decreasing of the level mapping from the head to the two first body atoms is to note that \( \text{Pre} \models \text{trans}(x, z, e, v) \) implies \(|e| \geq |v|\), and then:

\[ 2 \cdot |e| - |v| + 1 > |e|, |v|. \]

Let us see how the Call Pattern Theorem 5.2 (ii) helps us in showing that TRANS is non-floundering. Consider a negative literal \( \neg \text{member'}(X, V) \) selected along a LD-derivation for TRANS and any query \( Q \) such that \( \vdash \{\text{Pre}\} Q \{\text{Post}\} \) holds by \( |\_| \).

By Theorem 5.2 (ii), \( \text{Pre} \models \text{member'}(X, V) \). Due to the form of \( \text{Pre} \), this implies that \( X \in z \) and \( V \in \text{List}(z) \). This and the fact that \( z \) is a set of constants imply that \( \neg \text{member'}(X, V) \) is ground. In particular, we have that TRANS is non-floundering.
w.r.t. $\text{Pre}$. By Theorem 5.6, we conclude that $\text{TRANS}$ is totally correct w.r.t. the specification $(\text{Pre}, \text{Post} \cap \text{Pre})$.

### 5.2.3. Completeness of LDNF-resolution

In the following, we present some results on completeness of Negation as Failure for LDNF-resolution, as a by-result of the verification method developed so far.

**Theorem 5.7 (Completeness of Negation as Failure).** Assume that $\vdash \{\text{Pre}\} \ P \ \{\text{Post}\}$ holds and $\text{comp}(P)$ is consistent. If for $A \in \text{Pre}$, $P \cup \{A\}$ does not flounder, then

1. if $\text{comp}(P) \models \neg A$ then there exists a finitely failed LDNF-tree for $P$ and $A$.
2. if $\text{comp}(P) \models A$ then there exists a LDNF-refutation for $P$ and $A$.

**Proof.** By the Termination Theorem 5.3 the LDNF-tree for $P$ and $A$ is finite. Since $P \cup \{A\}$ is non-floundering either the LDNF-tree is finitely failed or there is a refutation.

1. The latter case is not possible, otherwise by Soundness of SLDNF-resolution ([41], Theorem 15.6) $\text{comp}(P) \models A$. This is in contradiction with the hypothesis $\text{comp}(P) \models \neg A$ and the assumption that $\text{comp}(P)$ is consistent.
2. The former case is not possible, otherwise by soundness of negation as failure ([41], Theorem 15.4) $\text{comp}(P) \models \neg A$. This is in contradiction with the hypothesis $\text{comp}(P) \models A$ and the assumption that $\text{comp}(P)$ is consistent. \[ \square \]

We are now in the position to give a declarative interpretation of $M^f_\text{p}$ and $\text{FF}^g_\text{p}$.

**Theorem 5.8.** Assume that $\vdash \{\text{Pre}\} \ P \ \{\text{Post}\}$ holds, $P$ is non-floundering and $\text{comp}(P)$ is consistent. Then:

1. $M^f_\text{p} \cap \text{Pre} = \{A \in \text{Pre} \mid \text{comp}(P) \models A\}$
2. $\text{FF}^g_\text{p} \cap \text{Pre} = \{A \in \text{Pre} \mid \text{comp}(P) \models \neg A\}$.

**Proof.** The $\subseteq$ inclusions follow from soundness of SLDNF-resolution and of the negation as failure rule. The $\supseteq$ inclusions follow from the Completeness Theorem 5.7. \[ \square \]

The next result extends completeness to ground general queries. Proof relation $\vdash$ naturally extends to general queries by discarding the $k > |A_i|$ requirements in Definition 5.1.

**Theorem 5.9 (Completeness of LDNF-resolution I).** Assume that $\vdash \{\text{Pre}\} \ P \ \{\text{Post}\}$ and $\vdash \{\text{Pre}\} \ Q \ \{\text{Post}\}$ hold, where $Q$ is a ground general query. Moreover, assume that $P \cup \{Q\}$ does not flounder and $\text{comp}(P)$ is consistent. If $\text{comp}(P) \models Q$ then there exists a LDNF-refutation for $P$ and $Q$.

**Proof.** The proof is by induction on the number of literals in $Q$.

1. **(Base)** If $Q$ consists of only one literal then the result follows by Theorem 5.7.
2. **(Step)** If $Q = L, Q'$ then by Theorem 5.7 there exists a LDNF-refutation for $L$. By Lemma 5.1 (iii, iv) we have $\text{Post} \models L$ and then, by Definition 5.3, $\vdash \{\text{Pre}\} \ Q' \ \{\text{Post}\}$
holds. Therefore we can apply the inductive hypothesis on \( Q' \) to reach the desired conclusion. □

The final result of this section is concerned with a further extension of completeness of LDNF-resolution. In this case, assuming an underlying language with infinitely many function symbols (i.e., \( \Sigma_L \) infinite), we can state a completeness result that extends a well-known theorem by Cavedon [21].

**Theorem 5.10 (Completeness of LDNF-resolution II).** Assume that \( \vdash_t \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) and \( \vdash \{ \text{Pre} \} \ Q \ \{ \text{Post} \} \) hold. Moreover, assume that \( P \cup \{ Q \} \) does not flounder, \( \Sigma_L \) is infinite and \( \text{comp}(P) \) is consistent. If \( \text{comp}(P) \models Q' \) for an instance \( Q' \) of \( Q \), then there exists a LDNF-refutation for \( P \) and \( Q \) with computed instance more general than \( Q' \).

**Proof.** Let \( Q'' \) be the query obtained by substituting every variable \( x_i \) in \( Q' \) by a term \( t_i \) with principal functor not appearing in \( P \) or \( Q' \), and distinct from that of the others \( x_j \), for \( j \neq i \). Such terms exist since \( \Sigma_L \) is infinite. \( P \cup \{ Q'' \} \) cannot flounder, otherwise by substituting the \( t_i \)'s with the \( x_i \)'s in the derivation, we would conclude that \( P \cup \{ Q' \} \) flounders, and a fortiori that \( P \cup \{ Q \} \) flounders. Therefore, by Theorem 5.9, there exists a LDNF-refutation for \( P \) and \( Q'' \). By substituting the \( t_i \)'s with the \( x_i \)'s along that refutation we obtain a LDNF-refutation for \( P \) and \( Q' \). Since \( P \cup \{ Q \} \) does not flounder, we can lift that refutation to a LDNF-refutation for \( P \) and \( Q \) with computed instance more general than \( Q' \). □

As a special case we find again the results of Cavedon [21] on acyclic programs. We recall that a program is acyclic if there exists a level mapping \( | \cdot | \) such that:

for every \( A \leftarrow L_1, \ldots, L_n \in \text{ground}_{L,L}(P) \) for \( i \in [1,n] \) \( |A| > |L_i| \),

where for a negative literal \( [\neg B_i] \) is set to \( |B_i| \).

It can be shown that if a program is acyclic with respect to a language \( L \), then it is acyclic with respect to every extension of \( L \). Therefore, we can assume, without loss of generality, that \( \Sigma_L \) is infinite.

Apt and Bezem ([1], Theorem 2.5) show that for an acyclic program \( P, M^L_P \) is a model of \( \text{comp}(P) \), i.e. that \( T_M(M^L_P) = M^L_P \). By Definition 5.3, we conclude that if \( P \) is acyclic then \( \vdash \{ B_L \} \ P \ \{ M^L_P \} \) holds in some language \( L \) with \( \Sigma_L \) infinite. In addition, \( \text{comp}(P) \) is consistent and \( \vdash \{ B_L \} \ Q \ \{ M^L_P \} \) holds for any query.

Summarizing, the only hypothesis needed to apply Theorem 5.10 is that \( P \cup \{ Q \} \) does not flounder, which is implied by the only hypothesis of the completeness theorem ([21], Theorem 4.5) that \( P \) and \( Q \) are allowed.

6. Related work


In the following, we discuss the relations of our approach with other proof methods for reasoning on (weak) partial correctness, termination, (weak) total correctness and general programs. Once again, we recall that our intended objective is to show that the proposed method – based on the \( \vdash \) relation – is a trade-off between expressiveness (i.e., the class of programs and properties it is able to reason about) and ease of use in paper & pencil verification proofs. In fact, it is apparent in the above references that the state-of-the-art in this area is that of a wide collection of separated methods and techniques, whose common issues are not properly recognized, and synthesized in a few unifying principles. Ours is an attempt towards this direction.

6.1. Weak partial correctness

Early works on proving declarative properties of logic programs can be traced back to Clark [22] and Hogger [37,38]. Apt and Marchiori [10] compare several methods, by showing how many of the proposal present in the literature adopt a Hoare's logic proof style [31], where specifications are given in terms of pre- and post-conditions. In particular, we refer the reader to Naish [44] for a paper investigating the parallel between verification of logic and imperative programs. Among the others, the method of Bossi and Cocco [16] is a trade-off between expressiveness and ease of use, being able to reason on declarative and run-time properties of Prolog programs. By allowing monotonic assertions only, i.e. assertions closed under substitution, they strictly extend the methods of:

- well-typed programs by Bronsard et al. [20], where directional types are considered to model the input/output behaviour of programs.
- well-moded programs by Dembinski and Maluszynski [27].

On the other hand, Apt and Marchiori [10] show that the method of Bossi and Cocco is a special case of:

- the inductive assertion method of Drabent and Maluszynski [32], which allows for the use of non-monotonic assertions, that is assertions not necessarily closed under substitution.
- the method of Colussi and Marchiori [23], where assertions are associated to control points, rather than to the relations defined in programs.

Another relevant approach is due to Deransart [28], who proposes two proof methods for weak partial correctness which are correct and complete. The first one is based on inductive specifications, i.e. specification that hold for the head of a clause if they hold for the body of the clause. The second one is a refinement of the first method in order to facilitate correctness proofs. Deransart [28], Section 6) points out that the method due to Bossi and Cocco is a special case of his proposals. Other recent approaches investigate extensions of pure logic programming including:

- declarative extensions of first-order built-in's of Frolog (Apt et al. [11]).
- methods to prove correctness with respect to the \( \mathcal{G} \)- and \( \mathcal{F} \)-semantics (Apt et al. [9]).
- general logic program (Ferrand and Deransart [36], Malfon [42]).
- concurrent constraint logic programs (de Boer et al. [25]).
- meta-programming (Pedreschi and Ruggieri [48]).
• dynamic scheduling systems (Apt and Luitjes [4], de Boer et al. [26]).

We now show that the method based on the relation $\vdash$ is equivalent with the one of Bossi and Cocco, thus precisely classifying the expressiveness of $\vdash$. We follow the presentation of Apt and Marchiori [10].

**Definition 6.1.** A type $I$ is a set of atoms such that if an atom $A$ is in $I$ then every instance of $A$ is in $I$. Let $pre, post$ be types. A program is well-m-asserted by $pre, post$ if for every $A \leftarrow B_1, \ldots, B_n$ instance of a clause from it, for $i \in [1, n]$:

$$A \in pre \land B_1, \ldots, B_{i-1} \in post \Rightarrow B_i \in pre$$

and

$$A \in pre \land B_1, \ldots, B_n \in post \Rightarrow A \in post.$$ 

A query is well-m-asserted by $pre, post$ if for every $B_1, \ldots, B_n$ instance of it, for $i \in [1, n]$:

$$A \in pre \land B_1, \ldots, B_{i-1} \in post \Rightarrow B_i \in pre.$$ 

A type $I$ is called *strongly monotonic* if an atom is in $I$ iff every ground instance of it is in $I$.

As an example of non strongly monotonic type, we mention the set of ground atoms. The method of Bossi and Cocco is based on proving a program well-m-asserted. It is evident that relation $\vdash$ is a simplification of well-m-assertedness. Intuitively, $\vdash$ coincides with well-m-assertedness restricted to strongly monotonic types. However, we claim that under a rather general hypothesis, the two methods exhibit the same expressiveness, in a sense clarified by the following definition.

**Definition 6.2.** Consider two sets of types, $\mathcal{I}$ and $\mathcal{J}$. We say that $\mathcal{I}$ is at least as expressive as $\mathcal{J}$ if every program well-m-asserted by two types $pre, post$ in $\mathcal{J}$ is well-m-asserted by $pre', post'$ types in $\mathcal{J}$ such that:

$$pre \subseteq pre' \quad \text{and} \quad post' \cap pre \subseteq post.$$ 

We say that $\mathcal{I}$ is as expressive as $\mathcal{J}$ if $\mathcal{J}$ is at least as expressive as $\mathcal{I}$ and vice versa. $\square$

In other words, $\mathcal{I}$ is at least as expressive as $\mathcal{J}$ if whenever we can reason on $P$ and $Q$ using types from $\mathcal{J}$, then we are able to reason on $P$ using types from $\mathcal{I}$ that allow for reasoning on a class of queries containing $Q$, since $pre \subseteq pre'$, and on finer properties, since for $post' \cap pre \subseteq post$.

**Theorem 6.1.** Assume that $\Sigma_L$ is infinite. Then strongly monotonic types are as expressive as types.

**Proof.** Obviously, types are at least as expressive as strongly monotonic types. Conversely, consider types $pre$ and $post$, and a program $P$ well-m-asserted by $pre, post$. We define the strongly monotonic types:

$$pre' = True(pre \cap B_L) \quad \text{and} \quad post' = True(M_p^l \cap post).$$
where \( \text{True}(I) = \{ A \in \text{Atom}_L \mid I \models A \} \). It is readily checked that \( P \) is well-m-asserted by \( \text{pre}', \text{post}' \) as well. Moreover, consider \( A \in \text{pre} \). Since every ground instance of \( A \) is in \( \text{pre} \cap B_L \), then \( A \) is in \( \text{pre}' \). Therefore, \( \text{pre} \subseteq \text{pre}' \).

In addition, if \( A \in \text{post}' \cap \text{pre} \) then every ground instance of \( A \) is in \( M'_P \). Let \( A' \) be a ground instance of \( A \) obtained by instantiating every variable of \( A \) with ground terms whose principal function symbol is distinct and does not appear in \( A \) or \( P \). \( A' \) exists since \( \Sigma_L \) is infinite. Then:

\[
A' \in M'_P
\]

\[
\iff \{ \text{A' ground} \}
\]

\[
P \models A'
\]

\[
\iff \{ \text{Theorem on Constants (see e.g. Ref. [52])} \}
\]

\[
P \models A.
\]

By Corollary 4.8 in Ref. [10], \( A \in \text{pre} \) and \( P \models A \) imply \( A \in \text{post} \), hence \( \text{post}' \cap \text{pre} \subseteq \text{post} \).

\[\square\]

6.2. Call pattern characterization

As shown in Section 3.3, the method based on the proof relation \( \vdash \) is not complete with respect to the notion of (weak) partial correctness, in the sense that there are programs \( P \) weak partially correct w.r.t. a specification \( \{ \text{Pre}, \text{Post}' \} \) for which \( \vdash \{ \text{Pre} \} P \{ \text{Post}' \} \) does not hold for any \( \text{Post}' \). A completeness result has been shown for the method of Deransart [28].

Incompleteness of relation \( \vdash \) is due to phenomena recently investigated by Boye and Malużynski [19]. They pointed out that directional types and, more generally, correctness can be viewed under two different aspects, depending whether one is interested in the input/output behaviour, i.e. declarative properties, or in the run-time behaviour of programs, i.e. call pattern characterization with respect to some specific selection rule. Under this distinction, the approach of Deransart is a method for proving declarative properties, while ours addresses also run-time properties with reference to the leftmost selection rule.

In particular, Definition 2.2 of weak partial and partial correctness is concerned with declarative properties, by noting that:

\[
M'_P \cap \text{Pre} \subseteq \text{Post}
\]

can be rewritten as

\[
M'_P \subseteq (B_L \setminus \text{Pre}) \cup \text{Post}.
\]

On the contrary, the proof method based on relation \( \vdash \) addresses call pattern characterization (see Corollary 3.1) as well as declarative properties. Therefore, we had to trade completeness of the method for the possibility of reasoning on call patterns with respect to the leftmost selection rule.

Naish [45] discusses the notion of types as supersets of the least Herbrand model in the sense of (5), by arguing that purely declarative information can actually express the essence of types and modes. He proposes [43,45] a definition of declarative typing of programs and applies it to program verification and type
checking. The resulting declarative proof method is more general than relation \( \vdash \) but still incomplete.

### 6.3. Partial correctness

Among the cited approaches to weak partial correctness, only Malfon [42] shows a method for proving that a postcondition is the strongest one, in the sense of Theorems 3.12 and 3.20. To the best of our knowledge, no approach discusses methods for characterizing the weakest (liberal) preconditions, in the sense of Theorem 3.21.

### 6.4. Termination

Concerning termination, we refer the reader to the survey of De Schreye and Decorte [51] for a comprehensive bibliography. Among others, Bezcm [14] and Apt and Pedreschi [13] introduced recurrent and acceptable logic programs, which are special cases of the proof relation \( \vdash \). In particular, a program \( P \) is acceptable iff \( \vdash \{ B_L \} P \{ Post \} \) holds for some \( Post \). and \( P \) is recurrent iff \( \vdash \{ B_L \} P \{ B_L \} \) holds.

Thus, our method can be viewed as an adaptation of the above universal termination proof methods with respect to the intended queries; this facilitates in many examples the required reasoning, in that uninteresting input queries are not to be taken into account. As an example, consider the following program \( FLAT \):

1. \( flat([], []). \)
2. \( flat([X | Xs], [f(X) | FXs]) \leftarrow flat(Xs, FXs). \)
3. \( flat(nil, []). \)
4. \( flat(tree(X, Ls, Rs), [f(X) | Fs]) \leftarrow flat(Ls, FLs), flat(Rs, FRs), append([], FSs, Fs). \)

augmented with the \( APPEND \) program. \( FLAT \) applies \( f(.) \) to every element of a given list, or of a preorder traversal of a given binary tree. We denote by \( Btree \) the set of binary trees, and for \( bt \in BTree \), \( \| bt \| \) denotes the number of nodes of \( bt \). Given:

- \( Pre = flat(GList \times U_L) \cup flat(BTree \times U_L) \cup Pre_{APPEND}. \)
- \( Post = \{ flat(ls, rs) \mid ls, rs \in GList \land \| ls \| = \| rs \| \} \cup \{ flat(bt, rs) \mid bt \in BTree. rs \in GList \land \| bt \| = \| rs \| \} \cup Post_{APPEND}. \)

it is straightforward to exhibit proof outlines to show that \( \vdash \{ Pre \} FLAT \{ Post \} \) holds by using a level mapping \( \| \) such that:

\[
\| flat(ls, rs) \| = \begin{cases} \| ls \| + 1 & \text{if } ls \in GList. \\ \| rs \| + 1 & \text{if } ls \in BTree. 
\end{cases}
\]

On the contrary, proving acceptability is awkward, due to the fact that badly-typed atoms have to be considered in the definition of the level mapping, such as \( tree(a.[a,b,c].tree(a.[ ].nil)) \).
Even worse, there are some interesting programs which terminate on a strict subset of $B_L$ only, and then cannot be acceptable. The most immediate example is the following program $\text{TRANSP}$:

$$\begin{align*}
\text{trans}(x, y, e) & \leftarrow x \sim_e y \text{ for a DAG } e \\
\text{trans}(X, Y, E) & \leftarrow \\
& \text{member}([X, Y], E).
\end{align*}$$

$$\begin{align*}
\text{trans}(X, Y, E) & \leftarrow \\
& \text{member}([X, Z], E), \\
& \text{trans}(Z, Y, E).
\end{align*}$$

augmented by the definition of $\text{member}$. In the intended meaning of the program, $\text{trans}(x, y, e)$ succeeds if $x \sim_e y$, i.e. if $[x, y]$ is in the transitive closure of a direct acyclic graph (DAG) $e$, which is represented as a list of pairs of constants. It is readily checked that if $e$ is not a DAG, i.e. it contains a cycle, then infinite derivations may occur. As a consequence, $\text{TRANSP}$ is not acceptable. Notice, however, that in the intended use of the program, $e$ is supposed to be a DAG. In our approach, we model that case by defining:

$$\begin{align*}
\text{Pre} & = \{ \text{trans}(x, y, e) \mid e \text{ is a DAG } \} \cup \{ \text{member}(x, Is) \mid \text{Is is a list}\} \\
\text{Post} & = \{ \text{trans}(x, y, e) \mid x \sim_e y \} \cup \{ \text{member}(x, Is) \mid x \text{ is in Is } \}.
\end{align*}$$

It is readily checked that $\vdash \{\text{Pre}\} \text{ TRANSP } \{\text{Post}\}$ holds by using the level mapping

$$\begin{align*}
|\text{trans}(x, y, e)| & = |e| + 1 + \text{Card}\{ z \mid x \sim_e z \} \\
|\text{member}(x, e)| & = |e|,
\end{align*}$$

where $\text{Card}$ is the set cardinality operator. By means of the same level mapping, it is readily checked that:

$$\begin{align*}
\vdash \{\text{Pre}\} \text{ trans}(x, Y, e) \{\text{Post}\} \text{ and } \vdash \{\text{Pre}\} \text{ trans}(X, Y, e) \{\text{Post}\}
\end{align*}$$

hold, where $e$ is a DAG. By Theorem 3.15, the LD-trees for $\text{trans}(x, Y, e)$ and $\text{trans}(X, Y, e)$ are finite.

### 6.5. (Weak) total correctness

We claim that the sum of well-m-assertedness and acceptability is not as expressive as the method based on proof relation $\vdash_r$. On the one hand, by simply applying in turn well-m-assertedness and acceptability involves considering more proof obligations than establishing $\vdash_r$. On the other hand, the complications with proving acceptability highlighted in the example programs $\text{FLAT}$ and $\text{TRANSP}$ still continue to hold. Furthermore, consider $P$ well-m-asserted by $\text{pre, post}$. Since in general $\text{post}$ is not a model of the program (see APPEND for an example), acceptability must be shown by considering a further set $\text{Post}'$ -- a model of $P$ -- which is not present in our approach. In addition, confusion can arise due to the fact that acceptability analysis acts at a ground level, whilst well-m-assertedness acts at a non-ground level.

Also, we mention that well-m-assertedness has been extended by Bossi et al. [17] to reason on termination. They define level mappings $\mid \mid$ on non-ground atoms as well, and require that for every $A \leftarrow B_1, \ldots, B_n$ instance of a clause of $P$, for every $i \in [1, n]$:
However, this leads to complications, since termination can be proved only using rigid level mappings, and then a further proof obligation has to be satisfied. $| |$ is called rigid if whenever $pre \models A$ then $|A| = |A'|$ for every instance $A'$ of $A$. Moreover, the resulting proof method is not complete in the sense of Theorem 3.19. In fact, consider the program $P$:

$$
p(0).
p(1) \leftarrow p(0).
$$

and $pre = \text{Atom}_l$, and any $post$. For every level mapping $| |$, we have that (6) requires $|p(1)| > |p(0)|$. Therefore, $| |$ cannot be rigid, since $pre \models p(X)$. On the contrary, it is straightforward to show that $\vdash \{B_t\} P \{B_t\}$ and $\vdash \{B_t\} p(X) \{B_t\}$ hold by the same level mapping. A similar argument applies to the termination proof method proposed for well-typed programs by Bresnard et al. [20].

6.6. General programs

Ferrand and Deransart [36] extend the proof method of Deransart [28] to prove declarative properties of general logic programs. Differently from our approach, they do not discuss termination issues and adopt the well-founded semantics [2]. As in the case of definite programs, their method is more general for proving declarative properties, albeit ours is also able reason on call pattern characterization and termination as well as ensuring completeness of LDNF-resolution.

The same arguments apply to the proposal of Malfon [42], which presents a correct and complete method to prove declarative properties with respect to Fitting and well-founded semantics [2]. It is worth noting that the notion of well-supported interpretation results to be a simplification of a similar notion introduced in [42].

6.7. Integrated approaches

There are a few attempts to present in a uniform way methods dealing with correctness, termination, call patterns, occur-check freedom, modular proofs, and other program properties. A valuable approach is due to Apt [8]. However, his book presents several separated results, which in many cases are instantiations of the proof method presented in this paper.

Also, Deville [30] proposes an approach for systematically deriving terminating programs from specifications provided in a Clark’s completion-like format. However, the method is not applicable to check correctness of existing programs.

Recently, Stärk [53] proposed a logic program theorem prover in which termination and correctness can be formally proved for programs containing negation and arithmetic built-in’s. The formal theory underlying the theorem prover is an extension of pure Prolog including induction principles and axioms for built-in’s.

7. Conclusions

The starting point of the research reported in this paper has been the recognition of a few core principles, common to several existing proof methods for logic pro-
grams. On this basis, a thorough proof theory has been developed as a candidate unifying framework capable of addressing a reasonably large spectrum of properties for a reasonably large class of programs.

The original contribution of this paper is the introduction of a proof relation \( \vdash \), for total correctness of logic programs, possibly containing negation and arithmetic built-in's, which are designed to be executed according to a fixed selection rule. In particular, the proposed proof theory concentrates on the (Prolog's) leftmost selection rule. For reasons of presentation, the \( \vdash \) proof method has been introduced in an incremental way, by a stepwise definition of increasingly higher levels of verification, from a weak form of partial correctness up to full-fledged total correctness.

Some applications of the method have been surveyed, including proving absence of run-time errors, modular program development, safe omission of the occur-check, verification of meta-programs, semantics decidability. By lack of space, we could not include the presentation of case studies of significant dimension. However, we refer the reader to [47] for a collection of case studies.

Finally, we compared the expressiveness of the proposed approach with existing proposals. Technically speaking, the proof theory is obtained as a combination of the proof method of Bossi and Cocco [16] for weak partial correctness, and the proof method of Apt and Pedreschi [13] for termination. The advantage of this operation is that the expressiveness of the combined method strictly exceeds the expressiveness of the separated methods both from a theoretical and a practical perspective.

We were not concerned here with the issue of automation, since the main focus was on the theoretical framework and, in addition, there would be no space in the paper for a fair accounting of the issue. However, we are pursuing a research line towards the design and the implementation of tools supporting systematic program development and automatic verification. Other interesting extensions we are currently investigating include constraint logic programs and dynamic selection rules.

Appendix A. Termination

In the following, we assume that the function \( \max : 2^N \rightarrow N \cup \{\infty\} \) is defined as follows:

\[
\max S = \begin{cases} 
0 & \text{if } S = \emptyset, \\
n & \text{if } S \text{ is finite and non-empty, and } n \text{ is the maximum of } S, \\
\infty & \text{if } S \text{ is infinite}.
\end{cases}
\]

Then \( \max S < \infty \) iff the set \( S \) is finite.

Moreover, we will use the finite multiset ordering. A multiset on \( W \) is an unordered sequence of elements from \( W \). We denote a multiset of elements \( a_1, \ldots, a_n \) by \( \text{bag}(a_1, \ldots, a_n) \). If \( W \) is associated with a irreflexive ordering \( < \), we define the ordering \( \prec_m \) on finite multiset induced by \( < \) as the transitive closure of the relation:

\[
X \prec Y \text{ iff } X = Y - \{a\} \cup Z \text{ for an } a \in Y \\
\text{and } Z \text{ such that } b < a \text{ for every } b \in Z.
\]
where $X, Y, Z$ are finite multiset of elements from $W$. A well-known result (see e.g. Ref. [29]) shows that if $(W, <)$ is a well-founded ordering, then the corresponding multiset ordering is well-founded as well. Here, we make use of an equivalent formulation of relation $\vdash$, for queries.

**Definition A.1.** Consider a level mapping $\mid$ and a specification $(\text{Pre}, \text{Post})$.

- To a query $Q = B_1, \ldots, B_n$ we associate $n$ sets of natural numbers defined as follows. For $i \in [1..n]$:
  
  $$|Q_i| = \{ |A_i||A_1, \ldots, A_n\text{ is a ground instance of } Q \text{ and } \text{Post} \vdash A_1, \ldots, A_{i-1} \}.$$  

- A query $Q$ is called bounded (by $\mid$ and $\text{Post}$) iff $|Q_i|$ is finite, for every $i \in [1..n]$.

- For $Q = B_1, \ldots, B_n$ bounded, we define the multiset $|Q|$ of natural numbers as follows:
  
  $$|Q| = \text{hag}(\max |Q_1|, \ldots, \max |Q_n|).$$

Observe that if $\vdash \{\text{Pre}\} \{\text{Post}\}$ holds, then $Q$ is bounded (by $\mid$ and $\text{Post}$) if and only if $\vdash \{\text{Pre}\} \{\text{Post}\}$ holds (by using $\mid$). The advantage of using the notion of boundedness is that we concentrate on termination only, without taking into account correctness, which is considered in Theorem 3.8.

Proofs of the theorems of this Appendix resemble a standard way to proceed (see similar proofs by Apt and Pedreschi [5] for acceptable programs) and, for lack of space, are not reported here. They can be found in Appendix A of Ref. [47].

**Lemma A.1.** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping $\mid$. Let $Q'$ be a LD-resolvent of $P$ and $Q$. Then

(i) $Q'$ is bounded (by $\mid$ and $\text{Post}$), and

(ii) $|Q'| \preceq_m |Q|$, and

(iii) $\vdash \{\text{Pre}\} Q' \{\text{Post}\}$.

The following result states a form of completeness, useful in proving the Termination Completeness Theorem 3.19.

**Theorem A.1.** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ and $P$ is Pre-terminating. Then there exists a level mapping $\mid$ such that:

(i) $\vdash \{\text{Pre}\} P \{\text{sp}(P, \text{Pre})\}$, and

(ii) for every query $Q$ such that $\vdash \{\text{Pre}\} Q \{\text{Post}\}$, we have that
  
  $\vdash \{\text{Pre}\} Q \{\text{sp}(P, \text{Pre})\}$
  
  holds by means of $\mid$ iff every LD-derivation for $P$ and $Q$ is finite.

Appendix B. General programs

**Lemma B.1.** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds. For every $A \in \text{Pre}$:

(i) If the LDNF-tree for $P$ and $A$ is finitely failed then $\text{Post} \vdash \neg A$.

(ii) If there is a LDNF-refutation for $P$ and $A$ then $\text{Post} \vdash A$.

**Proof.** The proof is by a double induction on the rank ($\geq 0$) and on the depth ($\geq 1$) of the finitely failed LDNF-tree in the case (i). In the case (ii), induction is on the rank ($\geq 0$) and on the length ($\geq 1$) of the LDNF-refutation.

(rank = 0):
(depth/length = 1) (i) If \( \text{Post} \models A \) then by Definition 5.3 (ii) there exists \( A \leftarrow L_1, \ldots, L_n \in \text{ground}_l(P) \) such that

\[ \text{Post} \models L_1, \ldots, L_n. \]

However, this is impossible since depth = 1 implies that \( A \) does not unify with any clause head.

(ii) Since length = 1, the hypothesis implies that \( A \) is an instance of the head of a unit clause. By Definition 5.3 (ii), we conclude \( \text{Post} \models A. \)

(depth/length > 1) (i) If \( \text{Post} \models A \) then by Definition 5.3 (ii) there exists \( C0 = A \leftarrow L_1, \ldots, L_n \in \text{ground}_l(P) \), with \( C \) clause from \( P \), such that

\[ \text{Post} \models L_1, \ldots, L_n. \]

Since the resolvent of \( A \) and \( C \) has a finitely failed LD(NF)-tree, every instance of its, and in particular \( L_1, \ldots, L_n \), has a finitely failed LD(NF)-tree. Therefore there exists \( i \in [1, n] \) such that \( L_1, \ldots, L_{i-1} \) have a refutation and \( L_i \) has a finitely failed LD(NF)-tree. Since rank = 0, \( L_1, \ldots, L_i \) are positive literals. By inductive hypothesis (ii) on the length of refutations:

\[ \text{Post} \models L_1, \ldots, L_{i-1} \]

and then, since \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \), \( \text{Pre} \models L_i \). By inductive hypothesis (i) on the depth \( \text{Post} \models L_i \). This contradicts (1), thus we conclude \( \text{Post} \models \neg A. \)

(ii) Consider the LDNF-resolvent of \( A \). Since rank = 0, every literal in it is positive. Moreover, some ground instance \( B_1, \ldots, B_n \) of it has a LD(NF)-refutation. By inductive hypothesis on the depth, we have

\[ \text{Post} \models B_1, \ldots, B_n. \]

Since \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \) holds, by Definition 5.3 (i) this implies \( \text{Post} \models A. \)

(rank > 0):

(depth/length = 1) Analogous to the case rank = 0.

(depth/length > 1) (i) If \( \text{Post} \models A \) then by Definition 5.3 (ii) there exists \( C0 = A \leftarrow L_1, \ldots, L_n \in \text{ground}_l(P) \), with \( C \) clause from \( P \), such that

\[ \text{Post} \models L_1, \ldots, L_n. \]

Since the resolvent of \( A \) and \( C \) has a finitely failed LD(NF)-tree, every instance of its, and in particular \( L_1, \ldots, L_n \), has a finitely failed LD(NF)-tree. Therefore there exists \( i \in [1, n] \) such that \( L_1, \ldots, L_{i-1} \) have a refutation and \( L_i \) has a finitely failed LD(NF)-tree. By inductive hypothesis (ii) on the length of refutations and (i) on the rank:

\[ \text{Post} \models L_1, \ldots, L_{i-1} \]

Since \( \vdash \{ \text{Pre} \} \ P \ \{ \text{Post} \} \), we have \( \text{Pre} \models L_i \), where \( L_i = A_i \) or \( L_i = \neg A_i \). We distinguish now two cases.

If \( L_i = A \), then by inductive hypothesis (i) on the depth \( \text{Post} \not\models L_i \). This contradicts (2), thus we conclude \( \text{Post} \models \neg A. \)

If \( L_i = \neg A \), then \( A \) has a LDNF-refutation with lower rank. By inductive hypothesis (ii) on the rank, we have \( \text{Post} \models A_i \), and then \( \text{Post} \not\models L_i \). This contradicts (2), thus we conclude \( \text{Post} \models \neg A. \)

(ii) Consider the resolvent of \( A \). We observe that some ground instance \( L_1, \ldots, L_n \) of it has a LDNF-refutation. By inductive hypothesis (ii) on the length of refutations every positive literal in \( L_1, \ldots, L_r \) is in \( \text{Post} \). By inductive hypothesis (i) on the rank, we have that every negative literal is true in \( \text{Post} \), i.e.
Since $\vdash \{\text{Pre}\} P \{\text{Post}\}$ holds, by Definition 5.3 (ii), this implies $\text{Post} \models A$. □

We extend the notion of boundedness to general queries by defining for a general query $Q$:

$$|Q| = \{ \{|A_i| \mid L_1, \ldots, L_n \text{ is a ground instance of } Q, \text{Post} \models L_1, \ldots, L_{i-1}, \text{ and } L_i = A_i \lor L_i = \neg A_i \} \}.$$

**Lemma B.2.** Assume that $\vdash \{\text{Pre}\} P \{\text{Post}\}$ and $\vdash \{\text{Pre}\} Q \{\text{Post}\}$ hold by the same level mapping $|\ |$. Let $Q'$ be a LDNF-resolvent of $P$ and $Q$. Then

(i) $Q'$ is bounded (by $|\ |$ and Post), and

(ii) $|Q'| \preceq_m |Q|$, and

(iii) $\vdash \{\text{Pre}\} Q' \{\text{Post}\}$.

**Proof.** In the case that a positive literal is selected, we follow the same reasoning of Lemma A.1. Therefore, we have only to consider $Q = \neg A, Q'$. In this case, $A$ is ground, and there exists a finitely failed LDNF-tree for $P$ and $A$, and $Q'$ is the LDNF-resolvent of $P$ and $Q$. By Lemma B.1 (i), $\text{Post} \models \neg A$. From this, (i–iii) readily follow. □

**References**


