Università di Pisa

Dipartimento di Informatica

Technical Report : TR-97-05

Verification of Logic Programs

Dino Pedreschi, Salvatore Ruggieri

February 10, 1997

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

Verification of Logic Programs

Dino Pedreschi and **Salvatore Ruggieri**

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail: {pedre, ruggieri}@di.unipi.it

February 10, 1997

Abstract

We propose a proof method in the style of Hoare's logic, aimed at providing a unifying framework for the verification of logic and Prolog programs with respect to their specifications. The method, which relies on purely declarative reasoning, has been designed as a trade-off between expressive power 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, total correctness, absence of run-time errors, safe omission of the occur-check, computed answers, modular program development. We finally generalize the method to general programs.

Keywords: Verification of logic and Prolog programs, Semantics of logic programs, Systematic program development, Hoare's logic.

Contents

1 Introduction **1**

2 Declarative Programming **3**
2.1 Which Semantics for Program Verification? 3
2.2 Specifications and Semantics 4

3 Proof Theory **6**
3.1 The Proof Method 6
3.2 Relating Correctness Proof Methods 11
3.3 Modularity 13
3.4 Weak Partial Correctness 17
3.5 Partial Correctness 18
3.6 Weak Total Correctness 23
3.7 Total Correctness 23
3.8 A Calculus of the Weakest Preconditions 26
3.9 Correct and Computed Instances 27

4 Applications **29**
4.1 Arithmetic Built-in's 29
4.2 Modular Verification 32
4.3 The Occur-check Problem 34
4.4 A Complete Example 36
4.5 Meta-interpreters and Meta-variables 37
4.6 Language Extension 39
4.7 Decidability Issues 42

5 General Logic Programs **43**
5.1 LDNF^{-}-resolution 43
5.2 LDNF-resolution 44

6 Conclusions **52**

A Termination **55**

B General Programs **59**

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 occur 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:

- partial correctness,
- termination,
- absence of type and run-time errors,
- safe omission of the occur-check,
- modular program development.

However, no comprehensive framework has been proposed, capable of addressing the various verification issues on the basis of a few simple, unifying principles. 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 [10].)

This paper introduces a proof theory designed as a candidate unifying framework for the verification of logic programs. The proof theory is based on Hoare's style triples $\{Pre\}\ P\ \{Post\}$ 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, and the selection rules, to mention few. In this sense, the proposed verification method is carefully designed as a compromise between generality and expressiveness from 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 [14], and the proof method for termination of Apt and Pedreschi [11]. The advantage of this combination is that the resulting method is applicable to a larger class of programs and properties, and with simpler proofs.

Related Work

Formal methods for reasoning about logic programs have been studied for a long time. De Schreye and Decorte [48] survey termination methods, Apt and Marchiori [9] survey modes, types, and partial correctness methods. Ducasseé and Noyé [28] survey environments for dynamic analysis and debugging. Crnogorac et al. [23] compare

some occur-check analysis methods. Also, Deransart [24] discusses and compares various verification approaches. Apt's book [7] collects several results on verification of Prolog programs. Deville [26] proposes an approach for systematically deriving (terminating) programs from specifications provided in a Clark's completion-like format. 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.

Among early references on correctness, we cite Clark and Tärnlund [20], Clark [19] and Hogger [31]. More recently, Drabent and Maluszynski [27] proposed a method for properties expressed using non-monotonic assertions, that is assertions not closed under substitution. Monotonic assertions are, instead, used by Bossi and Cocco [14]. In Section 3.2, we specifically relate our proof method with the one of Bossi and Cocco. In all these approaches, specifications are associated with the relations occurring in the program. In Colussi and Marchiori [21] an alternative proof method is introduced, in which assertions are attached to program points. Apt et al. [8] give methods to prove correctness with respect to the computed answer semantics. Pedreschi and Ruggieri [43] apply the verification method presented in this paper to meta-programming. Moreover, many of the approaches have been refined to deal with general programs, arithmetic built-in's and modular program development.

Concerning termination, early works are by Naish [39], Vasak and Potter [53], Ullmann and van Gelder [52] and Plümer [44]. Bezem [13] and Apt and Pedreschi [11] introduced *recurrent* and *acceptable* logic programs, which are special cases of our proof method, as shown in Section 3.2. We also cite transformational approaches by Baudinet [12], and Wang and Shyamasundar [54], and approaches based on abstract interpretation by De Schreye et al. [49].

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 correctness, termination, modular development, and computed instances of intended queries. In Section 4, we discuss refinements of the proof method and applications to real programs. In particular, we show how to reason on arithmetic built-in's, modular verification, safe omission of the occur check, verification of meta-interpreters and decidability issues. Finally, in Section 5, verification of general logic programs is considered. We extend the proof method to handle negation, and as a by-product we obtain a form of completeness of LDNF-resolution.

Notation

A word on terminology is in order. Throughout the paper we use the standard notation of logic programming, as in [6, 34], unless specified otherwise. A language 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 [32]. For the moment, we only assume the restricted case of *atoms-as-terms* ambivalent syntax. In Section 4.5, we also consider the *terms-as-atoms* case. 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. We use queries instead of goals. LD(NF)-resolution is SLD(NF)-resolution with the leftmost selection rule.

$L_P = \langle \Sigma_P , \Pi_P \rangle$ is the language generated by the program P. By M_P^L we denote the

least Herbrand model of P with L as the underlying language. M_P is an abbreviation for $M_P^{L_P}$. B_L is the Herbrand base of L. B_P is an abbreviation for B_{L_P}. We denote with $[A]_L$ the set of ground instances on the language L of the atom A. FF_P^L is the finite failure set of P, and FF_P^c is $B_L \setminus FF_P^L$.

We write $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ 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 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: $T_P(I) = \{ A \in B_L \mid \exists A \leftarrow B_1, \ldots, B_n \in ground_L(P) \ : \ B_1, \ldots, B_n \in I \}$.

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.

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;
- if the final letter of an identifier is "s", such as xs, Xs, then the identifier is intended to denote a list;
- symbols and expressions from the underlying language L are written in the *typewriter* style, such as `member(x), X, Xs, Y`;
- bold identifiers, such as **B** and **C**, are intended to denote a sequence of zero or more atoms.

With this assumptions, x is a ground term, X a term, **X** a sequence of atoms, and `X` a logic (or Prolog) variable. The list-length function is defined as follows: $|f(\ldots)| = 0$ if $f \neq$ `[.|.]`, and $|$`[`$X|T$`]`$| = |T| + 1$.

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^L (also known as $\mathcal{M}$-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. [29] (also known as the least term model of Clark [19]) and the $\mathcal{S}$-semantics of Falaschi et al. [30, 16].

DEFINITION 2.1 For a logic program P we define

$$\begin{aligned} \mathcal{M}(P) &= \{ A \in B_L \mid P \models A \} \\ \mathcal{C}(P) &= \{ A \in Atom_L \mid P \models A \} \\ \mathcal{S}(P) &= \{ A \in Atom_L \mid A \text{ is a computed instance of a pure atom} \}. \end{aligned}$$ □

In [8], 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 $\mathcal{M}$-semantics. Under

certain conditions, the $\mathcal{C}$- and $\mathcal{S}$-semantics can be reconstructed starting from the least Herbrand model. Therefore, $\mathcal{M}$-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^L 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. The justification of this statement is based on two considerations.

On the one hand, specifications generally deal with top level procedures only, leaving unspecified the description of auxiliary procedures. This is the case in top-down program development and modular programming.

On the other hand, the absence of types implies that the least Herbrand model is generally polluted with ill-typed atoms. Consider the `APPEND` program and its specification taken from [51]:

`append(`xs`, `ys`, `zs`)` $\leftarrow$ zs is the concatenation of the lists xs and ys.

```
append( [], Xs, Xs ).
append( [X|Xs], Ys, [X|Zs] ) ←
   append( Xs, Ys, Zs ).
```

`APPEND` is intuitively correct with respect to its specification but (if there are sufficiently many symbols in the language) its *intented interpretation* is not a model of the program. In fact, in the least Herbrand model ill-typed 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. In this paper, we adopt an approach to logic program verification which is based on the *intended* or *well-typed* fragment of the least Herbrand models [45]. To a certain extent, this approach resembles the Naish [40] statement *Specification = Program + Types*.

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 initial states of the admissible input, and postconditions specify (properties of) the final states of 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 $M_P^L \cap Pre$.

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 $M_P^L \cap Pre = Post$.
- P is *totally correct* w.r.t. a specification $(Pre, Post)$ iff $M_P^L \cap Pre = Post$ and $Pre \subseteq M_P^L \cup FF_P^L$, where FF_P^L is the finite failure set of P.

In addition, P is *weak* partially or totally correct if the weaker requirement $M_P^L \cap Pre \subseteq Post$ holds. □

As a consequence of this definition, partial and total correctness of a program P w.r.t. a specification $(Pre, Post)$ entails that $Post$ coincides with the well-typed fragment of M_P^L, namely with the *correct instances* of the intended queries. As usual, the difference between partial and total correctness is that, in the latter case, we also require termination, namely that every query in Pre either succeeds or (finitely) fails. It should be noted that both partial and total correctness are defined in purely declarative terms, as the sets M_P^L and FF_P^L are constructed without reference to the procedural interpretation of logic programming [34, 7]. The notions of weak partial and partial correctness compare with the notions of *correctness* and *completeness*, respectively, used by Deransart [24]. Yet other different terminologies are used by other authors, such as Deville [26], to refer to similar notions.

As an example, the `APPEND` program is intuitively totally correct w.r.t. $(Pre, Post)$ where

$$\begin{aligned} Pre &= \{\, \texttt{append}(xs,\ ys,\ zs) \mid xs, ys \text{ are lists} \,\} \\ Post &= \{\, \texttt{append}(xs,\ ys,\ zs) \mid zs = xs * ys \,\}. \end{aligned}$$

where $*$ is the list concatenation operator. In the next section we introduce a proof theory for partial (and total) correctness which enables us to formalize the reasoning adopted in this example. In particular, our method allows us to abstract away from the ill-typed fragment of M_P^L.

Notice that the *weak* version of either notions of correctness entails that $Post$ specifies some property of $M_P^L \cap Pre$. For instance, the `APPEND` program is weak totally correct w.r.t. $(Pre, Post')$, where Pre is as above, and

$$Post' = \{\, \texttt{append}(xs,\ ys,\ zs) \mid |zs| = |xs| + |ys| \,\}$$

where $|.|$ denotes the length of a list. Therefore, a postcondition in the sense of the weak correctness describes a property of every correct instance of an intended query.

Moreover, observe that the condition $Pre \subseteq M_P^L \cup FF_P^L$, used to define (weak) total correctness, is equivalent to require the so-called *existential termination* w.r.t. a fair selection rule, namely that any fair SLD-tree either contains a refutation or is finitely failed. We will actually deal with *universal termination* with respect to the leftmost selection rule, namely Prolog's. This fact is immediately reflected by the definition of our proof method, in which a left-to-right ordering of clause bodies is taken into account. For this reason, the proposed proof theory for total correctness is not complete in this general sense, albeit it is complete in a weaker sense, which will be clarified later.

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 query Q such that $Pre \models Q$, namely any query true in Pre, or equivalently, any query whose instances are included in the precondition.

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 $\{Pre\}\ P\ \{Post\}$ (for programs P) and $\{Pre\}\ Q\ \{Post\}$ (for queries Q). Triples are the basic tools to prove correctness. The proof method provides two distinct relations $\vdash$ and $\vdash_t$ on triples. We shall show how the relation $\vdash$ allows us to reason about (weak) partial correctness, whereas the relation $\vdash_t$ allows us to reason about (weak) total correctness.

The next crucial definition explains the proof obligations needed to prove a triple. As a consequence of the fact that a specification $Pre, Post$ of a program is assigned in terms of sets of ground atoms, we reason directly about ground instances of program clauses and queries. We recall from [13] the notion of a *level mapping* (on a language L) which is a function $|\ |: B_L \rightarrow N$ of ground atoms to natural numbers. Level mappings play the role of termination functions in the proofs of total correctness.

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 and a query Q, and a specification $(Pre, Post)$. We write:

- $\vdash_t \{Pre\}\ P\ \{Post\}$ iff there exists a level mapping $|\ |$ such that for every $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$

  1. for $i \in [1, n]$
  $Pre \models A \ \wedge\ Post \models B_1, \ldots, B_{i-1} \Rightarrow$
  (a) $Pre \models B_i$ and
  (b) $|A| > |B_i|$

  2. $Pre \models A \ \wedge\ Post \models B_1, \ldots, B_n \Rightarrow Post \models A$

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

- $\vdash \{Pre\}\ Q\ \{Post\}$ iff for every ground instance $A_1, \ldots, A_n$ of Q

  3. for $i \in [1, n]$ $Post \models A_1, \ldots, A_{i-1} \Rightarrow Pre \models A_i$

- $\vdash_t \{Pre\}\ Q\ \{Post\}$ iff there exist a level mapping $|\ |$ and $k \in N$ such that for every ground instance $A_1, \ldots, A_n$ of Q

  4. for $i \in [1, n]$ $Post \models A_1, \ldots, A_{i-1} \Rightarrow Pre \models A_i \ \wedge\ k > |A_i|$ □

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

1. each atom B in the body of C is in Pre when the the head of C is in Pre and all the atoms to the left of B in the body of C are in $Post$;

2. the head of C is in $Post$ when it is in Pre and all the atoms in the body of C are in $Post$.

In the case of $\vdash_t \{Pre\}\ P\ \{Post\}$ the decreasing of the level mapping is also required. 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 Pre) when the atoms to its left have been already executed (i.e., they are in $Post$). However, the presented notion is purely declarative, and no procedural intuition is needed to carry on the proof obligations. Similar considerations apply to the notion of a triple for a query.

The relation $\vdash$ is devised for proving (weak) partial correctness, whereas relation $\vdash_t$ allows us to prove (weak) total correctness: this role of triples will be clarified later. Some considerations about the above definition are in order.

Proving that a triple is in the relation $\vdash$ or $\vdash_t$ 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*.

Intuitively speaking, a precondition characterizes the intended (or well-typed) queries. A ground instance of a clause whose head is not in the precondition is superfluous as we never use it in a ground derivation starting with an atom of the precondition. The level mapping plays the role of a termination function. Strictly speaking, the level mapping has to be defined only on the precondition.

We point out that Definitions 3.2 *(3,4)* for a query Q derive from Definitions 3.2 *(1,2)* for the program $Ans \leftarrow Q$ where Ans is a new predicate. This follows from the following useful relation, which is immediate from Definition 3.2 *(1,2)*. For $\vdash \{Pre\}\ P\ \{Post\}$ and $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$,

$$A \in Pre \quad \text{implies} \quad \vdash \{Pre\}\ B_1, \ldots, B_n\ \{Post\} \tag{1}$$

Example: preorder tree traversal

As an example to clarify the form of the needed proof obligations, consider the program `PREORDER`:

`preorder(`T`, `Ls`)` $\leftarrow$
Ls is a preorder traversal of the binary tree T

(p1) `preorder(nil, [] ).`
(p2) `preorder(leaf(X), [X] ).`
(p3)
```
preorder(tree(X, Left, Right), Ls) ←
    preorder(Left, As),
    preorder(Right, Bs),
    append([X|As], Bs, Ls).
```

augmented with the `APPEND` program. The set of ground binary trees $Tree(\alpha, \beta)$ whose leaves belong to $\alpha \subseteq U_L$ and intermediate nodes belong to $\beta \subseteq U_L$ is defined by the grammar:

`Tree ::= nil | leaf(`α`) | tree(`β`, Tree, Tree)`

For instance, if $\alpha = \{0, 1, 2, \ldots\}$ and $\beta = \{+, -, *, \ldots\}$, we have that $Tree(\alpha, \beta)$ 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:

$$\|\texttt{nil}\| = 0$$
$$\|\texttt{leaf}(x)\| = 1$$
$$\|\texttt{tree}(x,\ l,\ l)\| = \|l\| + \|r\| + 1.$$

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

$$Pre_{\texttt{PREORDER}} = \{\texttt{preorder}(t,\ ls)\ \mid t \in Tree(\alpha,\beta)\} \cup Pre_{\texttt{APPEND}}$$

or, in a more intuitive representation:

$$Pre_{\texttt{PREORDER}} = \texttt{preorder}(Tree(\alpha,\beta) \times U_L) \cup \texttt{append}(List \times List \times U_L).$$

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 PREORDER, such as the meaningless atom `preorder(tree(0, leaf([]), leaf(nil)), [0,[],nil])`. Indeed, a complete characterization of $M^L_{\texttt{PREORDER}}$ is mucj more laborious than simply reasoning on correctness and termination of atoms in $Pre_{\texttt{PREORDER}}$.

A candidate level mapping is:

$$|\texttt{preorder}(t,\ ls)| = \|t\| + 1$$
$$|\texttt{append}(as,\ bs,\ cs)| = |as|$$

for atoms belonging to $Pre_{\texttt{PREORDER}}$ and 0 otherwise. 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 *intended interpretation* of PREORDER:

$Post_{\texttt{PREORDER}} = \{\ \texttt{preorder}(t,\ ls)\ \mid t \in Tree(\alpha,\beta) \wedge ls$ is a preorder traversal of $t\ \}$.

We are now in the position to prove $\vdash_t \{Pre_{\texttt{PREORDER}}\}$ PREORDER $\{Post_{\texttt{PREORDER}}\}$, by showing the proof obligation 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:

```
preorder(tree(x, left, right), ls)  ←
    preorder(left, as),
    preorder(right, bs),
    append([x|as], bs, ls).
```

Assume that the head is in the precondition, i.e.

$$x \in \beta \quad \text{and} \quad left, right \in Tree(\alpha,\beta).$$

By definition of $Pre_{\texttt{PREORDER}}$, also `preorder(`$left$`, `as`)` and `preorder(`$right$`, `bs`)` are in the precondition. Moreover,

$$\begin{aligned} & |\texttt{preorder}(\texttt{tree}(x,\ left,\ right),\ ls)| \\ = \ & \|left\| + \|right\| + 2 \\ > \ & \|left\| + 1 \\ = \ & |\texttt{preorder}(left,\ as)| \end{aligned}$$

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 $Pre_{\texttt{PREORDER}}$, i.e. that `[`x`|`as`]` and bs are lists, under the assumption that `preorder(`$left$`, `as`)` and `preorder(`$right$`, `bs`)` are in $Post_{\texttt{PREORDER}}$, which is obvious. Moreover,

$$\begin{aligned} & |\texttt{preorder(tree(}x\texttt{, }left\texttt{, }right\texttt{), }ls\texttt{)}| \\ = \;& \|left\| + \|right\| + 2 \\ > \;& \|left\| + 1 \\ = \;& |as| + 1 \\ = \;& |\texttt{append([}x\texttt{ | }as\texttt{], }bs\texttt{, }ls\texttt{)}| \end{aligned}$$

since `preorder(`$left$`, `as`)` $\in Post_{\texttt{PREORDER}}$ 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

`tree(`x`, `$left$`, `$right$`)` $\in Tree(\alpha, \beta)$, and
as is a preorder traversal of $left$, and
bs is a preorder traversal of $right$, and
$ls =$ `[`x` | `as`] * `bs

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

In conclusion, we have proved that $\vdash_t \{Pre_{\texttt{PREORDER}}\}$ `PREORDER` $\{Post_{\texttt{PREORDER}}\}$.

Lifting up to non-ground queries, we observe that

$$\vdash_t \{Pre_{\texttt{PREORDER}}\}\ \texttt{preorder(}t\texttt{,L)}\ \{Post_{\texttt{PREORDER}}\}$$

holds, where t is a (possibly non-ground) term with every ground instance in $Tree(\alpha, \beta)$, and `L` is a variable. In particular, if $\alpha = \beta = U_L$ then the nodes of the tree t can be any term. Analogously, if $\alpha = List(U_L)$ the leaf nodes can be any list. The query `preorder(`t`,L)` is intended to calculate a preorder traversal of a given tree t.

Proof Outlines

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

DEFINITION 3.3 A proof outline for a clause $A \leftarrow A_1, \ldots, A_n$ and $|\ |$, Pre, $Post$ is a labeled clause of the form

$$\begin{array}{lll} \{g_0\} & & \\ A_0 & \leftarrow & \{t_0\} \\ & \quad \{g_1\} & \\ & A_1, & \{t_1\} \\ & \quad \{f_1\} & \\ & \quad \{g_2\} & \\ & \vdots & \\ & A_{n-1}, & \{t_{n-1}\} \\ & \quad \{f_{n-1}\} & \\ & \quad \{g_n\} & \\ & A_n. & \{t_n\} \\ & \quad \{f_n\} & \\ \{f_0\} & & \end{array}$$

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:

(i) for $i \in [0,n]$: $t_i = |A_i|$,
(ii) for $i \in [0,n]$: $g_i \Leftrightarrow A_i \in Pre$,
(iii) for $i \in [0,n]$: $f_i \Leftrightarrow A_i \in Post$,
(iv) for $i \in [1,n]:\ g_0 \wedge f_1 \wedge \ldots \wedge f_{i-1} \Rightarrow g_i \wedge t_0 > t_i$,
(v) $g_0 \wedge f_1 \wedge \ldots \wedge f_n \Rightarrow f_0$. □

The intuition is that the assertion g_i specifies the conditions under which the atom A_i of the clause is in Pre, the assertion f_i specifies the conditions under which A_i is in $Post$, 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, we have that $\vdash_t \{Pre\}\ P\ \{Post\}$ if and only if there exists a level mapping $|\ |$ and a proof outline for each clause of P and $|\ |, Pre, Post$.

Let us see the proof outlines for `PREORDER`.

```
  { nil ∈ Tree(α,β) }
preorder(nil, [] ).        { 1 }
  { nil ∈ Tree(α,β) ∧  [] is a preorder traversal of it  }

  { leaf(X)∈ Tree(α,β) }
preorder(leaf(X), [X] ). { 2 }
  { leaf(X) ∈ Tree(α,β) ∧  [X] is a preorder traversal of it  }

  { tree(X, Left, Right) ∈ Tree(α,β) }
preorder(tree(X, Left, Right), Ls) ←          ||Left|| + ||Right|| + 2
     { Left ∈ Tree(α,β) }
   preorder(Left, As),                        ||Left|| + 1
     { Left ∈ Tree(α,β) ∧ As is a preorder traversal of it  }
     { Right ∈ Tree(α,β) }
   preorder(Right, Bs),                       ||Right|| + 1
     { Right ∈ Tree(α,β) ∧ Bs is a preorder traversal of it  }
     { [X|As] and Bs are lists  }
   append([X|As], Bs, Ls).                    |As| + 1
     { Ls = [X|As] * Bs }
  { tree(X, Left, Right) ∈ Tree(α,β) ∧
    Ls is a preorder traversal of it   }
```

The related proofs have been shown in the previous section.

Proof outlines become simpler when considering the relation $\vdash$. In the labeled clause no level mapping t_i appears and the proof obligations are (ii-iii-v) plus the following simplified version of (iv):

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

Again, by construction $\vdash \{Pre\}\ P\ \{Post\}$ holds iff there exists a proof outline for each clause of P and $Pre, Post$.
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.

In the sequel, we shall also make use of a more general form of proof outlines, which enable us to use assertions f_i and g_i which do not coincide with post and preconditions,

but rather represent strengthenings and weakenings of $Post$ and Pre, according to the following equivalent interpretation of the assertions used in proof outlines. We denote with h_i for $i \in [0, n+1]$ the assertion $g_0 \wedge f_1 \wedge \ldots \wedge f_{i-1}$. Thus, in particular, $h_0 = h_1 = g_0$.

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

(ii) $g_0 \Leftarrow A_0 \in Pre$, and for $i \in [1, n]$: $h_i \wedge g_i \Rightarrow A_i \in Pre$,

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

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

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

With this more liberal interpretation, simpler proof outlines can be constructed. The construction of a proof outline satisfying the proof obligations (i) through (v) for every clause in a program P amounts to showing that $\vdash_t \{Pre\}\ P\ \{Post\}$ holds. In the following figure it is depicted how (i) through (iv) imply conditions *1(a)* and *1(b)* of Definition 3.2.

$$\begin{array}{lllllllllll} g_0 & \wedge & f_1 & \wedge & \ldots & \wedge & f_{i-1} & \Rightarrow & g_i & \wedge & t_0 > t_i \\ \Uparrow & & \Uparrow & & & & \Uparrow & & \Downarrow & & \Downarrow \\ A_0 \in Pre & \wedge & A_1 \in Post & \wedge & \ldots & \wedge & A_{i-1} \in Post & \Rightarrow & A_i \in Pre & \wedge & |A_0| > |A_i| \end{array}$$

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

3.2 Relating Correctness Proof Methods

In this section, we compare the expressiveness of relations $\vdash$ and $\vdash_t$ to other proof methods, including the partial correctness method of Bossi and Cocco [14] , and the termination proof method of Apt and Pedreschi [11].

Bossi and Cocco introduced an assertion-based method for proving correctness of logic programs. We recall it by substantially following the presentation of Apt and Marchiori [9].

DEFINITION 3.4 A type I is a set of atoms such that if an atom is in I then every its instance 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 $i \in [1, n]$

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

and

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

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

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

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

In the original paper [14] and in Apt and Marchiori [9] the actual definition of types is given in terms of a set of assertions in (an extension of) a first-order language. Therefore, relation $\vdash$ is a simplification of well-m-assertedness. In particular, $\vdash$ coincides with well-m-assertedness restricted to strongly monotonic types. In fact, let us define for a Herbrand interpretation J

$$True(J) = \{A \in Atom_L \mid J \models A\}$$

i.e. $True(J)$ is the set of true atoms in the interpretation J. With this notation, we have that a type I is strongly monotonic if it is of the form $True(J)$ for some J. The next result characterizes relation $\vdash$ as well-m-assertedness restricted to strongly monotonic types.

THEOREM 3.5 *$\vdash \{Pre\}\ P\ \{Post\}$ (resp., $\vdash \{Pre\}\ Q\ \{Post\}$) holds iff P (resp.,Q) is well-m-asserted by True(Pre), True(Post).* □

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 3.6 Consider two classes 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 $\mathcal{I}$ 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 $\mathcal{I}$ isat least as expressive as $\mathcal{J}$ and vice versa. □

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 $A \in pre$ we have $post \subseteq post'$.

Although relation $\vdash$ is a simplification of well-m-assertedness, involving reasoning on ground instances only, we show that, under the hypothesis that the underlying Σ_L is infinite, the two methods have the same expressiveness. In fact, in this situation strongly monotonic types are as expressive as types.

THEOREM 3.7 *Assume that the underlying language L contains infinitely many function symbols, i.e. that Σ_L is infinite. Then strongly monotonic types are as expressive as types.*

Proof. Obviously, types are at least as expressive as strongly monotonic types, since they are a superset of them.

Conversely, consider types pre and $post$, and a program P well-m-asserted by $pre, post$. We define

$$pre' = True(pre \cap B_L) \quad post' = True(M_P^L \cap post).$$

It is easy to see that P is well-m-asserted by $pre', post'$ as well.

Moreover, for every A, if $A \in pre$ then every ground instance of A is in pre and then A is in pre'. Therefore, $pre \subseteq pre'$.

On the other hand, if $A \in post' \cap pre$ then every ground instance of A is in M_P^L. 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 L has infinitely many function symbols. Then

$$\begin{array}{ll} & A' \in M_P^L \\ \Leftrightarrow & \quad \{A' \text{ ground } \} \\ & P \models_L A' \\ \Leftrightarrow & \quad \{ \text{ Theorem on Constants (see e.g. [50]) } \} \\ & P \models_L A \end{array}$$

By [9, Corollary 4.8], $A \in pre$ and $P \models_L A$ implies $A \in post$.
Therefore $post' \cap pre \subseteq post$, which concludes the proof. □

Concerning relation $\vdash_t$, we observe that it is related to the termination proof method introduced by Apt and Pedreschi [11], namely to *acceptable* programs. In particular, P is acceptable iff $\vdash \{B_L\}\ P\ \{Post\}$ holds for some $Post$. Also, $\vdash_t$ extends the approach introduced by Bezem [13] defining *recurrent* programs. It results that P is recurrent iff $\vdash \{B_L\}\ P\ \{B_L\}$ holds. Thus, our method can be viewed as a relativization of the above 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. The method of Bossi and Cocco has been extended in [15] to reason on termination. They define level mappings on non-ground atoms as well. However, this leads to complications, since correctness can be proved only using a restricted class of level mappings, called *rigid*, which limits the expressiveness of the method.

In conclusion, the proof method based on relation $\vdash_t$ combines the approaches of [14] and [11] into a unified theoretical framework addressing at the same time the various verification issues, with the advantage of making proofs simpler.

3.3 Modularity

Proving $\vdash \{Pre\}\ P\ \{Post\}$ and $\vdash_t \{Pre\}\ P\ \{Post\}$ 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

$$\vdash \{Pre\}\ P \cup Q \{Post\}$$

holds, starting from triples for P and Q.
Here we consider how to prove $\vdash \{Pre\}\ P\ \{Post\}$ starting from triples involving simpler specifications. First, we show that the following (*Hoare's logic* style) rule is valid:

THEOREM 3.8

$$\frac{\vdash \{Pre\}\ P\ \{Post\} \quad \vdash \{Pre\}\ P\ \{Post'\}}{\vdash \{Pre\}\ P\ \{Post \cap Post'\}}.$$

Proof. Let $A \leftarrow B_1, \ldots, B_n$ be a ground instance of a clause from P:

- for $i \in [1, n]$, if $Pre \models A$ and $Post \cap Post' \models B_1, \ldots, B_{i-1}$ then $\vdash \{Pre\}\ P\ \{Post\}$ implies $Pre \models B_i$;
- if $Pre \models A$ and $Post \cap Post' \models B_1, \ldots, B_n$ then $\vdash \{Pre\}\ P\ \{Post\}$ implies $Post \models A$ and $\vdash \{Pre\}\ P\ \{Post'\}$ implies $Post' \models A$. Therefore $Post \cap Post' \models A$.

□

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:

$$\{\ \texttt{sort}(xs,\ ys) \mid\ ys \text{ is an ordered permutation of } xs\ \}$$

then we can split it into:

$$\{\ \texttt{sort}(xs,\ ys) \mid\ ys \text{ is a permutation of } xs\ \}$$

and

$$\{\ \texttt{sort}(xs,\ ys) \mid\ ys \text{ is an ordered list }\}.$$

Proving separately the correctness of the two simpler postconditions generally involves less efforts than proving the correctness of their conjunction.

On the other hand, Theorem 3.8 allows us to define the notion of *strongest postcondition*.

DEFINITION 3.9 Assume that $\vdash \{Pre\}\ P\ \{Post\}$ holds. We denote by $sp(P, Pre)$ the intersection of every $Post'$ such that $\vdash \{Pre\}\ P\ \{Post'\}$, and call it the *strongest postcondition* of P and Pre. □

It is worth noting that the strongest postcondition is defined only for programs for which there exists at least one $Post$ such that $\vdash \{Pre\}\ P\ \{Post\}$. In fact, there exist programs and preconditions for which $\vdash \{Pre\}\ P\ \{Post\}$ does not hold for any $Post$, such as $\mathtt{p} \leftarrow \mathtt{q}$ and $Pre = \{\mathtt{p}\}$.

Symmetrically, the following rule holds which allows to simplify the precondition.

THEOREM 3.10

$$\frac{\vdash \{Pre\}\ P\ \{Post\} \quad \vdash \{Pre'\}\ P\ \{Post\}}{\vdash \{Pre \cup Pre'\}\ P\ \{Post\}}.$$

Proof. Let $A \leftarrow B_1, \ldots, B_n$ be a ground instance of a clause from P:

- for $i \in [1, n]$, if $Pre \cup Pre' \models A$ and $Post \models B_1, \ldots, B_{i-1}$ then $Pre \models A$ or $Pre' \models A$. In both cases, either $\vdash \{Pre\}\ P\ \{Post\}$ or $\vdash \{Pre'\}\ P\ \{Post\}$ imply $Pre \models B_i$ or $Pre' \models B_i$. Therefore, $Pre \cup Pre' \models B_i$;
- if $Pre \cup Pre' \models A \wedge Post \models B_1, \ldots, B_n$ then $Pre \models A$ or $Pre' \models A$. In both cases, either $\vdash \{Pre\}\ P\ \{Post\}$ or $\vdash \{Pre'\}\ P\ \{Post\}$ imply $Post \models A$.

□

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

DEFINITION 3.11 We denote by $wlp(P, Post)$ the union of every Pre' such that $\vdash \{Pre'\}\ P\ \{Post\}$, and call it the *weakest liberal precondition* of P and $Post$. □

The next result shows that $sp(P, Pre)$ and $wlp(P, Post)$ are respectively a postcondition and a precondition.

THEOREM 3.12

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

□

We next extend Theorems 3.8 and 3.10 to relation $\vdash_t$.

THEOREM 3.13

$$\frac{\vdash_t \{Pre\}\ P\ \{Post\} \quad \vdash_t \{Pre\}\ P\ \{Post'\}}{\vdash_t \{Pre\}\ P\ \{Post \cap Post'\}}.$$

Proof. Suppose that $\vdash_t \{Pre\}\ P\ \{Post\}$ holds using the level mapping $|\ |_1$, and $\vdash_t \{Pre\}\ P\ \{Post'\}$ holds using $|\ |_2$. Then we show that $\vdash_t \{Pre\}\ P\ \{Post \cap Post'\}$ holds using $|\ |_1 + |\ |_2$ (or equivalently $min\{|\ |_1, |\ |_2\}$). By Theorem 3.8, we have only to show the decreasing of the level mapping. Consider $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$:

- for $i \in [1,n]$, if $Pre \models A \wedge Post \cap Post' \models B_1, \ldots, B_{i-1}$ then, by hypothesis, $|A|_1 > |B_i|_1$ and $|A|_2 > |B_i|_2$. Therefore

$$|A|_1 + |A|_2 > |B_i|_1 + |B_i|_2$$

and analogously for min. □

Notice that it is not needed to define a notion of strongest *terminating* postcondition, as the intersection of all the $Post$ such that $\vdash_t \{Pre\}\ P\ \{Post\}$. In fact, $\vdash_t \{Pre\}\ P\ \{Post\}$ trivially implies $\vdash_t \{Pre\}\ P\ \{sp(P, Pre)\}$, 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 $\vdash_t \{Pre\}\ P\ \{Post\}$. We start by stating the following rule:

THEOREM 3.14

$$\frac{\vdash_t \{Pre\}\ P\ \{Post\} \quad \vdash_t \{Pre'\}\ P\ \{Post\}}{\vdash_t \{Pre \cup Pre'\}\ P\ \{Post\}}$$

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

$$\|A\| = \begin{cases} min\{|A|_1, |A|_2\} & \text{if } A \in Pre \cap Pre' \\ |A|_1 & \text{if } A \in Pre \setminus Pre' \\ |A|_2 & \text{if } A \in Pre' \setminus Pre \end{cases}$$

and $\|\ A\| = 0$ otherwise, we show that $\vdash_t \{Pre \cup Pre'\}\ P\ \{Post\}$ holds using the level mapping $\|\ \|$. By Theorem 3.10, we have only to show the decreasing of the level mapping. Consider $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$:

- for $i \in [1,n]$, if $Pre \cup Pre' \models A \wedge Post \models B_1, \ldots, B_{i-1}$ then we distinguish three cases:
  1. if $Pre \cap Pre' \models A$ then, by hypothesis, $|A|_1 > |B_i|_1$ and $|A|_2 > |B_i|_2$. Therefore
  $$min\{|A|_1, |A|_2\} > min\{|B_i|_1, |B_i|_2\}$$
  2. if $Pre \setminus Pre' \models A$ then $\vdash_t \{Pre\}\ P\ \{Post\}$ implies $B_i \in Pre$ and then $|A|_1 > |B_i|_1 \geq min\{|B_i|_1, |B_i|_2\}$. Therefore, in both cases $B_i \in Pre'$ and $B_i \notin Pre'$, we have a decreasing of the level mapping;
  3. if $Pre' \setminus Pre \models A$ the reasoning is symmetric to the previous case. □

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

$$\begin{aligned} |\texttt{append}(xs,\ ys,\ zs)|_1 &= |xs| \\ Pre' &= \texttt{append}(List(\alpha) \times List(\alpha) \times U_L) \\ Post &= \{\ \texttt{append}(xs,\ ys,\ zs)\ \mid zs = xs * ys\ \} \end{aligned}$$

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 `append(Xs, Ys,` zs`)` binds `Xs` to a prefix and `Ys` to a suffix of zs. Formally, one can show that $\vdash_t \{Pre''\}$ `APPEND` $\{Post\}$ holds by the level mapping $|\ |_2$, where

$$\begin{aligned} |\texttt{append}(xs,ys,\ zs)|_2 &= |zs| \\ Pre'' &= \texttt{append}(U_L \times U_L \times List(\alpha)) \end{aligned}$$

Then, by Theorem 3.14, we conclude that $\vdash_t \{Pre' \cup Pre''\}$ `APPEND` $\{Post\}$ holds by means of the following level mapping:

$$|\texttt{append}(xs,\ ys,\ zs)| \quad = \quad \begin{cases} min\{|xs|,|zs|\} & \text{if } xs, ys, zs \in List(\alpha) \\ |xs| & \text{if } xs, ys \in List(\alpha), zs \notin List(\alpha) \\ |zs| & \text{if } xs, ys \notin List(\alpha), zs \in List(\alpha) \\ 0 & \text{otherwise} \end{cases}$$

The next definition introduces the notion of a weakest precondition.

DEFINITION 3.15 We denote by $wp(P, Post)$ the union of every Pre' such that $\vdash_t \{Pre'\}\ P\ \{Post\}$, and call it the *weakest precondition* of P and $Post$. □

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

$$\texttt{p} \leftarrow \texttt{p}.$$

and $Post = \{\ \texttt{p}\ \}$, we have that

$$wlp(P, Post) = \{\ \texttt{p}\ \} \quad \neq \quad \emptyset = wp(P, Post).$$

By generalizing the proof Theorem 3.14, we obtain that $wp(P, Post)$ is indeed a precondition for P and $Post$.

THEOREM 3.16

$$\overline{\vdash_t \{wp(P, Post)\}\ P\ \{Post\}}.$$

Proof. Consider the sequence $Pre_1, Pre_2, \ldots$ of all preconditions Pre_i such that $\vdash_t \{Pre_i\}\ P\ \{Post\}$ holds using $|\ |_i$. We define $\|A\| = min\{\ |A|_i\ :\ A \in Pre_i, i \geq 1\ \}$, for $A \in wp(P, Post)$, and $\|\ A\| = 0$ otherwise.
Let us show that $\vdash_t \{wp(P, Post)\}\ P\ \{Post\}$ holds using $\|\ \|$.
Consider $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$.

- Assume that, for $j \in [1, n]$, $wp(P, Post) \models A$ and $Post \models B_1, \ldots, B_{j-1}$. For every Pre_i such that $A \in Pre_i$, we have that $Pre_i \models B_j$ and $|A|_i > |B_j|_i$. Therefore, $wp(P, Post) \models B_j$ and $\|A\| = min\{\ |A|_i\ :\ A \in Pre_i, i \geq 1\ \} > min\{\ |B_j|_i\ :\ B_j \in Pre_i, i \geq 1\ \} = \|B_j\|$.
- If $wp(P, Post) \models A$ and $Post \models B_1, \ldots, B_n$, then there exists Pre_i such that $Pre_i \models A$ and $Post \models B_1, \ldots, B_n$. Therefore, $Post \models A$.

□

We conclude this section by introducing two further rules:

THEOREM 3.17

$$\frac{\vdash \{Pre\}\ P\ \{Post\}}{\vdash \{Pre\}\ P\ \{Post \cap Pre\}} \qquad \frac{\vdash_t \{Pre\}\ P\ \{Post\}}{\vdash_t \{Pre\}\ P\ \{Post \cap Pre\}}$$

Proof. Let $A \leftarrow B_1, \ldots, B_n$ be a ground instance of a clause from P:

- for $i \in [1, n]$, if $Pre \models A$ and $Post \cap Pre \models B_1, \ldots, B_{i-1}$ then,

  $$Post \models B_1, \ldots, B_{i-1}$$

  and, by hypothesis, $Pre \models B_i \quad (\ \wedge\ |A| > |B_i|)$;

- if $Pre \models A$ and $Post \cap Pre \models B_1, \ldots, B_n$ then, $Post \models B_1, \ldots, B_n$ and, by hypothesis, $Post \models A$. Therefore, $Post \cap Pre \models A$. □

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

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

3.4 Weak Partial Correctness

A key result is the following theorem, stating the persistency of the relation $\vdash$ along SLD-derivations.

THEOREM 3.18 (PERSISTENCY) *Assume that*

$$\vdash \{Pre\}\ P\ \{Post\} \qquad \textit{and} \qquad \vdash \{Pre\}\ Q\ \{Post\}\ .$$

Then for every SLD-resolvent Q' of P and Q,

$$\vdash \{Pre\}\ Q'\ \{Post\}.$$

Proof. First, we show that if $Q = A_1, \ldots, A_n$ is a ground query and $A_i \leftarrow B_1, \ldots, B_l \in ground_L(P)$ 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 $\vdash \{Pre\}\ Q'\ \{Post\}$ holds. Let us show the proof obligations:

- for $k \in [1, i-1]$, if $Post \models A_1, \ldots, A_{k-1}$ then $Pre \models A_k$ since $\vdash \{Pre\}\ Q\ \{Post\}$ holds;
- for $k \in [1, l]$, if $Post \models A_1, \ldots, A_{i-1}, B_1, \ldots, B_{k-1}$ then $Pre \models A_i$, since $Post \models A_1, \ldots, A_{i-1}$ and $\vdash \{Pre\}\ Q\ \{Post\}$ holds.
  This and $A_i \leftarrow B_1, \ldots, B_l \in ground_L(P)$ imply $Pre \models B_k$;
- for $k \in [i+1, n]$, if $Post \models A_1, \ldots, A_{i-1}, B_1, \ldots, B_l, A_{i+1}, \ldots, A_{k-1}$ then $Pre \models A_i$, since $Post \models A_1, \ldots, A_{i-1}$ and $\vdash \{Pre\}\ Q\ \{Post\}$ holds.
  This and $A_i \leftarrow B_1, \ldots, B_l \in ground_L(P)$ and $Post \models B_1, \ldots, B_l$ imply $Post \models A_i$, and then

  $$Post \models A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_{k-1}.$$

  This and $\vdash \{Pre\}\ Q\ \{Post\}$ imply that $Pre \models A_k$.

Consider now a not necessarily ground $Q = A_1, \ldots, A_n$, and let

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

be the resolvent of Q and a variant $B \leftarrow B_1, \ldots, B_l$ of a clause from P, where θ 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 $(B \leftarrow B_1, \ldots, B_l)\theta$. From the first part of this proof, we conclude that for every ground instance Q'' of Q', $\vdash \{Pre\}\ Q''\ \{Post\}$ holds. By Definition 3.2, this is equivalent to say that $\vdash \{Pre\}\ Q'\ \{Post\}$ holds, hence the thesis. □

As a special case, considering the leftmost selection rule, we have that for every selected atom A, $Pre \models A$. Another immediate consequence is that every computed

instance of P and Q is true in the interpretation $Post$. This is a formal counterpart to the intuitive notion that a postcondition is a description of corrected instances of intended queries.

COROLLARY 3.19 *Assume that* $\vdash \{Pre\}\ P\ \{Post\}$ *and* $\vdash \{Pre\}\ Q\ \{Post\}$. *Then*
(i) for every atom A *selected in a LD-derivation for* P *and* Q, $\quad Pre \models A$,
(ii) for every computed instance Q' *of* P *and* Q, $\quad Post \models Q'$.

Proof.
(i) Let A, Q' be a query in a LD-derivation for P and Q. By Theorem 3.18, we have that $\vdash \{Pre\}\ A, Q'\ \{Post\}$. By Definition 3.2, $Pre \models A'$ for every instance A' of A. Therefore $Pre \models A$.
(ii) Let $\mathtt{p}^n$ be a fresh predicate symbol, where $x_1, \ldots, x_n$ are the variables of Q. We define

$$Pre' = Pre \cup \mathtt{p}^n(U_L \times \ldots \times U_L)$$

$$Post' = Post \cup \{\mathtt{p}^n(t_1, \ldots, t_n) \mid Post \models Q\{x_i/t_i \mid i \in [1,n]\}\}.$$

With this assumptions, it is readily checked that $\vdash \{Pre'\}\ P\ \{Post'\}$, since $\mathtt{p}^n$ does not appear in P, and $\vdash \{Pre'\}\ Q,\ \mathtt{p}^n(\mathtt{X}_1, \ldots, \mathtt{X}_n)\ \{Post'\}$, by definition of $Post'$. 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 P and $Q, \mathtt{p}^n(\mathtt{X}_1, \ldots, \mathtt{X}_n)$ where $\mathtt{p}^n(t_1, \ldots, t_n)\theta$ is selected, with $Q\{x_i/t_i \mid i \in [1,n]\}$ variant of Q'. By definition of $Post'$, we conclude that $Post \models Q'$. □

As an example, consider a term T such that every ground instance of it is in $Tree(\alpha, \beta)$. We showed that $\vdash \{Pre_{\mathtt{PREORDER}}\}\ \mathtt{preorder(}T\mathtt{,\ X)}\ \{Post_{\mathtt{PREORDER}}\}$ holds. By Corollary 3.19, every ground instance of the computed instance $\mathtt{preorder(}T\mathtt{,}X\mathtt{)}$ is in $Post_{\mathtt{PREORDER}}$. In other words, for every ground instance of $\mathtt{preorder(}T\mathtt{,}X\mathtt{)}$, the second argument is a preorder traversal of the tree in the first argument. In general, we are in the position to state the Weak Partial Correctness theorem.

THEOREM 3.20 (WEAK PARTIAL CORRECTNESS) *If* $\vdash \{Pre\}\ P\ \{Post\}$ *holds then* P *is weak partially correct w.r.t. the specification* $(Pre, Post)$, *i.e.*

$$M_P^L \cap Pre \subseteq Post.$$

Proof. Consider $A \in M_P^L \cap Pre$. Then $\vdash \{Pre\}\ A\ \{Post\}$ holds, and by Strong Completeness of SLD-resolution there exists a LD-refutation for P and A. Therefore $A \in Post$ by Corollary 3.19 *(ii)*. □

3.5 Partial Correctness

The Weak Partial Correctness Theorem 3.20 suggests to investigate further the relation between postconditions and the well-typed fragment of the least Herbrand model, i.e. $M_P^L \cap Pre$. Since every postcondition is a superset of $M_P^L \cap Pre$, if we prove that $\vdash \{Pre\}\ P\ \{M_P^L \cap Pre\}$, then we conclude that $M_P^L \cap Pre$ is the strongest postcondition. This fact is made precise in the following theorem.

THEOREM 3.21 *Assume that* $\vdash \{Pre\}\ P\ \{Post\}$ *holds. Then* $\vdash \{Pre\}\ P\ \{M_P^L \cap Pre\}$ *holds, and consequently*

$$sp(P, Pre) = M_P^L \cap Pre.$$

Proof. Consider $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$.

- For $i \in [1,n]$, if $Pre \models A \wedge M_P^L \cap Pre \models B_1, \ldots, B_{i-1}$ then by Theorem 3.20, $Post \models B_1, \ldots, B_{i-1}$, and, by the hypothesis, $Pre \models B_i$.
- Suppose that $Pre \models A \wedge M_P^L \cap Pre \models B_1, \ldots, B_n$. As M_P^L is a model of P, we conclude that $M_P^L \models A$, and then $M_P^L \cap Pre \models A$.

Therefore $\vdash \{Pre\}\ P\ \{M_P^L \cap Pre\}$ holds.
By Definition 3.9 we have that $sp(P, Pre) \subseteq M_P^L \cap Pre$. The converse inclusion $sp(P, Pre) \supseteq M_P^L \cap Pre$ follows from Theorems 3.12 and 3.20. □

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

THEOREM 3.22 (PARTIAL CORRECTNESS) *If $\vdash \{Pre\}\ P\ \{Post\}$ holds then P is partially correct w.r.t. the specification $(Pre, sp(P, Pre))$.* □

The problem is now how to characterize the strongest postcondition directly, without first constructing the least Herbrand model M_P^L. To this end, we introduce the notion of a *well-supported interpretation*, introduced in the context of semantics for general logic programs (see [1]). An similar notion has been employed for program verification w.r.t. three-valued semantics by Malfon [35].

DEFINITION 3.23 Assume that $\vdash \{Pre\}\ P\ \{Post\}$ holds.
$Post$ is a *well-supported interpretation* (w.r.t. P and Pre) iff there exist

- a well-founded poset $(W, <)$, and
- a function $|\ | : B_L \rightarrow W$

such that for any $A \in Post \cap Pre$ there exists $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ such that

$$\forall i \in [1,n] : Post \models B_i \wedge |A| > |B_i|.$$ □

The idea underlying this definition is to require that any atom in $Post \cap Pre$ 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. Therefore, well-supportedness combined with the requirements of Definition 3.2 entails that indeed M_P^L coincides with a well-supported interpretation restricted on Pre. Observe that, in Definition 3.23, the condition $Post \cap Pre \models B_i$ for $i \in [1,n]$ is equivalent to $Post \models B_i$ for $i \in [1,n]$ under the assumption that $\vdash \{Pre\}\ P\ \{Post\}$ holds.

Summing up, the next theorem shows that well-supportedness yields a correct and complete proof method for partial correctness.

THEOREM 3.24 *Let P be a program such that $\vdash \{Pre\}\ P\ \{Post\}$. Then*

$$Post \cap Pre = sp(P, Pre) \quad \textit{iff} \quad Post \ \textit{is well-supported (w.r.t. } P \textit{ and } Pre\textit{).}$$

Proof.
($\Rightarrow$) We consider the naturals with the ordering relation $<$ as well-founded poset. Let $|\ | : B_L \rightarrow N$ be the function

$$|A| = \begin{cases} min\{i : A \in T_P \uparrow i\} & \text{if } A \in M_P^L \cap Pre \\ 0 & \text{otherwise.} \end{cases}$$

$|\ |$ is well-defined since $M_P^L = T_P \uparrow \omega$. By hypothesis, $Post \cap Pre = sp(P, Pre)$, and then, by Theorem 3.21, $Post \cap Pre = M_P^L \cap Pre = T_P \uparrow \omega \cap Pre$. To show the

conclusion, we prove by induction on $i \geq 0$ that for every $A \in T_P \uparrow i \cap Pre$ there exists $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ such that

$$\forall i \in [1,n] : Post \models B_i \wedge |A| > |B_i|.$$

The base case is trivial, since $T_P \uparrow 0 = \emptyset$. Let A be in $T_P \uparrow i \cap Pre$. If $A \in T_P \uparrow i-1$ then the conclusion follows from the inductive hypothesis. On the contrary, let A be in $T_P \uparrow i \backslash T_P \uparrow i-1 \cap Pre$. Then, there exists $A \leftarrow B_1, \ldots, B_n$ in $ground_L(P)$ such that

$$T_P \uparrow i-1 \models B_1, \ldots, B_n$$

and since $A \in Pre$ and $\vdash \{Pre\}\ P\ \{Post\}$

$$T_P \uparrow i-1 \cap Pre \models B_1, \ldots, B_n .$$

By Theorem 3.20, $Post \supseteq M_P^L \cap Pre \supseteq T_P \uparrow i-1 \cap Pre$ for every $i \geq 0$. Then

$$Post \models B_1, \ldots, B_n .$$

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

$$|A| = i > i-1 \geq |B_k|.$$

($\Leftarrow$)) We only show that $Post \cap Pre \subseteq M_P^L$, as the converse inclusion follows directly from Theorem 3.20. Consider $A \in Post \cap Pre$ and the maximal tree T such that:

- A is the root ;
- if B is a node such that $B \in Post \cap Pre$ and $B_1, \ldots, B_n$ are its sons then $B \leftarrow B_1, \ldots, B_n \in ground_L(P)$ and

$$Post \models B_1, \ldots, B_n \quad \wedge \quad |B| > max_{i=1}^n |B_i|.$$

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 $B \leftarrow B_1, \ldots, B_n$ such that $Post \models B_1, \ldots, B_n$ and then T is not maximal. As a conclusion, T is a proof tree for A (see [19]), and then $A \in M_P^L$. □

As a final observation, we point out that whenever $\vdash \{Pre\}\ P\ \{Post\}$ holds, $sp(P, Pre)$ is well-supported. In fact, by Theorem 3.21 $\vdash \{Pre\}\ P\ \{sp(P, Pre)\}$ holds, and $sp(P, Pre) \cap Pre = sp(P, Pre)$: thus, we are in the hypothesis of Theorem 3.24.

Proof Outlines

Proving weak partial correctness is handy, as the task can be carried out using the proof outlines. In contrast, Definition 3.23 may seem intricate and difficult to handle with simple tools. Fortunately, it has a straightforward interpretation in terms of proof outlines.

DEFINITION 3.25 A proof outline for a clause $A \leftarrow A_1, \ldots, A_n$, a function $|\ |$: $B_L \rightarrow W$ into a well-founded poset $(W, <)$ and Pre, $Post$, is a labeled clause of the form

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

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 $i \in [1,n]$: $g \wedge f_i \Rightarrow A_i \in Post$,

(iii) for $i \in [1,n]$: $g \Rightarrow f_i \wedge t_0 > t_i$. □

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 $A_i \in Post \wedge |A_0| > |A_i|$. 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 Pre we have to show that there exist proof outlines for (instances of) clauses from P and a same function $|\ | : B_L \rightarrow W$ 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.

Example: lexicographic ordering

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

```
(l1)   ≤( [X, Y], [X, Y] ).
(l2)   ≤( [X, Y], [s(U), V] ) ←
          ≤( [X, Y], [U, Y] ).
(l3)   ≤( [X, Y], [X, s(V)] ) ←
          ≤( [X, Y], [X, V] ).
```

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

$$Pre = \{\leq([a,b],[c,d]) \mid a,b,c,d \in N\ \}\ \}$$
$$Post = \{\leq([a,b],[c,d]) \mid a,b,c,d \in N, (a,b) <_{lex} (c,d)\}$$

where $\leq_{lex}$ is the lexicographic ordering relation.

We observe that the expected the recursive clauses are general than ($l2, l3$). A natural version of the program is the following:

```
(l1)   ≤( [X, Y], [X, Y] ).
(l4)   ≤( [X, Y], [s(U), V] ) ←
          ≤( [X, Y], [U, Z] ).
(l5)   ≤( [X, Y], [U, s(V)] ) ←
          ≤( [X, Y], [U, V] ).
```

Although *(l2, l3)* are instances of *(l4, l5)*, we are still in the position to show that `LEXORD` is partially correct w.r.t. the specification $(Pre, Post)$, by means of Theorem 3.24. First, we note that $\vdash \{Pre\}$ `LEXORD` $\{Post\}$. Next, we fix $W = (N \times N, \leq_{lex})$, and define $|\ |$ as follows:

$$|\leq([a,b],[c,d])| = (c-a, d-b))$$

where difference over natural numbers is interpreted as a total function, by letting $x - y = 0$ if $x < y$. The proof outlines for well-supportedness follow.

(l1) $\{\ a = c \ \wedge\ d = b\ \}$
`≤( [a, b], [a, b] ).` $\{\ (0,0)\ \}$

(l2) $\{\ a \leq c\ \}$
`≤( [a, b], [s(c), d] )` $\leftarrow$ $\{\ (c-a+1, d-b)\ \}$
`≤( [a, b], [c, b] ).` $\{\ (c-a, 0)\ \}$
$\{\ (a,b) \leq_{lex} (c,b)\}$

(l3) $\{\ a = c \ \wedge\ b \leq d\ \}$
`≤( [a, b], [a, s(d)] )` $\leftarrow$ $\{\ (0, d-b+1)\ \}$
`≤( [a, b], [c, d] ).` $\{\ (0, d-b)\ \}$
$\{\ (a,b) \leq_{lex} (a,d)\}$

Since the decreasing of $|\ |$ is obvious, we need only to show that every atom in $Pre_{\texttt{LEXORD}} \cap Post_{\texttt{LEXORD}}$ is an instance of some head, and that the associated assertion holds. Let $\leq([a,b],[c,d])$ in $Pre_{\texttt{LEXORD}} \cap Post_{\texttt{LEXORD}}$. Then $(a,b) \leq_{lex} (c,d)$.

By definition of $\leq_{lex}$, we have that $(a,b) \leq_{lex} (c,d)$ iff $a < c \vee (a = c \wedge b \leq d)$, i.e. iff $(a < c) \vee (a = c \wedge b = d) \vee (a = c \wedge b < d)$. Proof outlines for *(l1)*, *(l2)* and *(l3)* cover respectively $(a = c \wedge b = d)$, $(a < c)$ and $(a = c \wedge b < d)$.

Therefore, $Post$ is well-supported, and, finally, partial correctness follows from Theorems 3.24 and 3.22, since

$$Post = Post \cap Pre = sp(\texttt{LEXORD}, Pre) = M^L_{\texttt{LEXORD}} \cap Pre.$$

Example: preorder tree traversal

Another example is `PREORDER`, when it is used in reverse mode, to compute the trees having a given preorder traversal, i.e. with the following precondition:

$$Pre'_{\texttt{PREORDER}} = \{\texttt{preorder(}t\texttt{, }ls\texttt{)} \mid ls \in List(\alpha \cup \beta)\} \cup Pre_{\texttt{APPEND}}$$

One can easily show that $Post_{\texttt{PREORDER}}$ is well-supported w.r.t. `PREORDER` and $Pre'_{\texttt{PREORDER}}$. On the other hand, it is worth noting that the LD-tree for `PREORDER` and an intended query such as `preorder(X,` ls`)` is infinite. However, other search strategies, different from Prolog's, can ensure termination. For instance, the delay declaration

```
DELAY preorder(T, Ls) UNTIL nonvar(Ls)
```

prevents infinite computations on systems such as MU-Prolog and Gödel, by delaying the selection of an atom `preorder(`T`,` Ls`)` until Ls is not a variable. More generally, all the results presented so far hold for every logic program, without any assumption on the underlying control, i.e. the selection rule and search strategy. For this reasons, Deransart [24] call "declarative" the properties of weak partial and partial correctness, although he uses a different terminology.

3.6 Weak Total Correctness

On the other hand, relation $\vdash_t$ allows us to reason on universal termination w.r.t. the leftmost selection rule only. However, no assumption is made on the search strategy. It is immediate from Definition 3.2 that the search strategy does not play any role in the decreasing of the level mapping. These remarks are formalized by the following results on persistence of $\vdash_t$ and weak total correctness.

THEOREM 3.26 (PERSISTENCY AND TERMINATION) *Assume that*

$$\vdash_t \{Pre\}\ P\ \{Post\} \quad \textit{and quad} \vdash_t \{Pre\}\ Q\ \{Post\}.$$

Then for every LD-resolvent Q' of P and Q,

$$\vdash_t \{Pre\}\ Q'\ \{Post\}.$$

Moreover, every LD-tree of P and Q is finite.

Proof. The fact that $\vdash_t \{Pre\}\ Q'\ \{Post\}$ holds is shown in Appendix A, Lemma A.2 *(iii)*. Moreover, by Lemma A.2 *(ii)* there cannot be an infinite branch in a LD-tree for P and Q. Since a LD-tree is finitely branching, by König's Lemma, it is finite. □

An immediate consequence of this result and Theorem 3.20 is the Weak Total Correctness Theorem.

THEOREM 3.27 (WEAK TOTAL CORRECTNESS) *If $\vdash_t \{Pre\}\ P\ \{Post\}$ holds then P is weak totally correct w.r.t. the specification $(Pre, Post)$.* □

As an example, as a result of the proof that

$$\vdash_t \{Pre_{\texttt{PREORDER}}\}\ \texttt{PREORDER}\ \{Post_{\texttt{PREORDER}}\}$$

`PREORDER` is weak totally correct w.r.t. $(Pre_{\texttt{PREORDER}}, Post_{\texttt{PREORDER}})$. In addition, for every term T whose ground instances are in $Tree(\alpha, \beta)$, the LD-tree of `PREORDER` and `preorder(`T`,L)` is finite.

3.7 Total Correctness

First, we state the corresponding of Theorem 3.21 for relation $\vdash_t$. This fact allows us to use the results of Section 3.5, in a simplified form, as a tools for proving total correctness .

THEOREM 3.28

$$\frac{\vdash_t \{Pre\}\ P\ \{Post\}}{\vdash_t \{Pre\}\ P\ \{M_P^I \cap Pre\}.}$$

□

The proof is analogous to that of Theorem 3.21. An immediate consequence of Theorem 3.28 is that $\vdash_t \{Pre\}\ P\ \{Post\}$ implies the total correctness of P w.r.t. the specification $(Pre, sp(P, Pre))$.

THEOREM 3.29 (TOTAL CORRECTNESS) *If $\vdash_t \{Pre\}\ P\ \{Post\}$ holds, then P is totally correct w.r.t. the specification $(Pre, sp(P, Pre))$.*

Proof. By Theorem 3.28, $\vdash_t \{Pre\}\ P\ \{sp(P, Pre)\}$ holds, since $sp(P, Pre) = M_P^L \cap Pre$. From this and Theorem 3.27, we have that P is weak totally correct w.r.t. $(Pre, sp(P, Pre))$. Again by the relation $sp(P, Pre) = M_P^L \cap Pre$, we conclude that P is totally correct w.r.t. $(Pre, sp(P, Pre))$. □

In addition to total correctness, we show a form of completeness of the termination proof method associated with relation $\vdash_t$. First, we generalize the notion of terminating programs used in the literature (see e.g. [11]), by relativizing it w.r.t. the precondition.

DEFINITION 3.30 For a program P and a set $Pre \subseteq B_L$, we say that P is Pre-terminating iff for any $A \in Pre$ every LD-tree for P and A is finite. □

From the next Theorem, we have that, if $\vdash \{Pre\}\ P\ \{Post\}$ holds then P is Pre-terminating iff $\vdash_t \{Pre\}\ P\ \{sp(P, Pre)\}$ holds. In other words, the relation $\vdash_t$ is a complete termination proof method, restricted to the triples in $\vdash$. In this sense, the proof method proposed here is a proper generalization of the method by Apt and Pedreschi [11].

THEOREM 3.31 (TERMINATION COMPLETENESS) *If $\vdash_t \{Pre\}\ P\ \{Post\}$ holds then P is Pre-terminating. Conversely, $\vdash \{Pre\}\ P\ \{Post\}$ and P Pre-terminating imply $\vdash_t \{Pre\}\ P\ \{sp(P, Pre)\}$.*

Proof. The *if-part* follows from Theorem 3.28, whereas the *onlyif-part* is reported in Appendix A as Theorem A.4 *(i)*. □

In addition, restricting attention to the atoms in the precondition may facilitate the termination proofs, as will be pointed out in later examples.

Proof Outlines

The notion of well-supported interpretations becomes simpler when considering relation $\vdash_t$. In the particular case that $Post \subseteq Pre$, it boils down to the well-known notion of supported interpretation (see e.g. [6]). As the next Theorem shows, the proof obligations on the level mapping used to prove $\vdash_t \{Pre\}\ P\ \{Post\}$ implicitly satisfy the requirements of Definition 3.23. As a consequence, we obtain a simpler proof method.

THEOREM 3.32 *Assume that $\vdash_t \{Pre\}\ P\ \{Post\}$ holds. Then the following statements are equivalent:*

(i) $Post \cap Pre = sp(P, Pre)$,

(ii) $T_P(Post) \supseteq Post \cap Pre$,

(iii) *Post is well-supported (w.r.t. P and Pre.)*

Proof. *(i)* $\rightarrow$ *(ii)* and *(iii)* $\rightarrow$ *(i)* are shown in Theorem 3.24. Let us prove the *(ii)* $\rightarrow$ *(iii)*. We show that $Post$ is a well-supported interpretation by considering the level mapping $|\ |$ used to prove $\vdash_t \{Pre\}\ P\ \{Post\}$. Consider $A \in Post \cap Pre$.
As $T_P(Post) \supseteq Post \cap Pre$, there exists a clause $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ such that $Post \models B_1, \ldots, B_n$. Moreover, since $\vdash_t \{Pre\}\ P\ \{Post\}$, it is straightforward to prove by induction on the n that $|A| > |B_i|$ for every $i \in [1, n]$. Therefore $Post$ is a well-supported interpretation w.r.t. P and Pre. □

Proof outlines for well-supportedness in the case of total correctness are simpler than those in the case of partial correctness, under the assumption that $\vdash_t \{Pre\}\ P\ \{Post\}$ holds.

DEFINITION 3.33 A proof outline $\mathcal{PO}$ for a clause $A \leftarrow B_1, \ldots, B_n$ and Pre, $Post$, is a construct of the form

$$\begin{array}{lll} & \{g\} & \\ A & \leftarrow & \\ & A_1, & \\ & \quad \{f_1\} & \\ & \vdots & \\ & A_{n-1}, & \\ & \quad \{f_{n-1}\} & \\ & A_n. & \\ & \quad \{f_n\} & \end{array}$$

where f_i, g, for $i \in [0, n]$ are assertions (in some formal logic), such that every ground instance of the following *proof obligations* holds:
(i) for $i \in [1, n]$: $g \wedge f_i \Rightarrow A_i \in Post$,
(ii) for $i \in [1, n]$: $g \Rightarrow f_i$. □

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 $A_i \in Post$. The use of the auxiliary assertions f_i's is not strictly necessary, albeit useful in constructing proof outlines.

In order to prove that $Post$ is well-supported w.r.t. P and Pre we have to show that there exists a set of proof outlines for (instances of) clauses from P such that every atom A in $Post \cap Pre$ is a ground instance of some head atom in a proof outline, and that the assertion associated with the head holds.

Example: preorder tree traversal

As an example, we report the proof outlines for `PREORDER`, omitting those for `APPEND`.

```
  { true }
preorder(nil, [] ).

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

  { tree(X, Left, Right) ∈ Tree(α,β) ∧
    Ls = [X|As] * Bs  is a preorder traversal of it   ∧  |As| = ||Left||   }
preorder(tree(X, Left, Right), Ls) ←
   preorder(Left, As),
     { Left ∈ Tree(α,β) ∧ As  is a preorder traversal of it   }
   preorder(Right, Bs),
     { Right ∈ Tree(α,β) ∧ Bs  is a preorder traversal of it   }
   append([X|As], Bs, Ls).
     { Ls = [X|As] * Bs  }
```

The proof obligations of the last proof outline are fulfilled, noting that the preorder traversal of $Left$ coincides with the sublist of Ls, the preorder traversal of `tree(`X`,` $Left$`,` $Right$`)`, from position 2 to position $\|Left\| + 1$, i.e. with As, and analogously for $Right$ and Bs.

We check that every atom in $Post_{\texttt{PREORDER}} \cap Pre_{\texttt{PREORDER}}$ is an instance of the head of some clause for which the assertion associated with the clause holds. Assume that `preorder(`t`,` ls`)` is in $Post_{\texttt{PREORDER}} \cap Pre_{\texttt{PREORDER}}$. Then

- either t is `nil` and ls is `[]`. This case is covered by the first proof outline;
- or t is `leaf(`x`)` for some x and ls is `[`x`]`. This case is covered by the second proof outline;
- or t is `tree(`x`,` $left$`,` $right$`)` and ls is `[`x `|` as `]` $*bs$ where as is a preorder traversal of $left$ and bs is a a preorder traversal of $right$. In particular, $|as|$ do coincide with $\|left\|$, and then this case is covered by the third proof outline.

Noting that $Post_{\texttt{PREORDER}} \cap Pre_{\texttt{PREORDER}} = Post_{\texttt{PREORDER}}$, we conclude that `PREORDER` is totally correct w.r.t. the specification $(Pre_{\texttt{PREORDER}}, Post_{\texttt{PREORDER}})$.

3.8 A Calculus of the Weakest Preconditions

Differently from the case of the strongest postconditions, we were unable to provide a simple direct characterization of the weakest (liberal) preconditions. However, in [45, 42] we present a theory of the weakest (liberal) preconditions for logic programming, with the aim of describing $wlp(P, Post)$ and $wp(P, Post)$ as fixpoints of the following operator $\vartheta_{P,Post}$, which is defined on the lattice of Herbrand interpretations.

DEFINITION 3.34 For a program P and a set $Post \subseteq B_L$, we define the function on Herbrand interpretation

$$\begin{aligned} \vartheta_{P,Post}(I) \quad = \quad \{A \in B_L \mid \forall A \leftarrow B_1, \ldots, B_n \in ground_L(P) \; : \\ \forall i \in [1,n] : Post \models B_1, \ldots, B_{i-1} \Rightarrow I \models B_i \\ \wedge \; Post \models B_1, \ldots, B_n \Rightarrow Post \models A \; \} \end{aligned}$$

□

This operator reveals monotonic, albeit not continuos. In the following, the main results from [45, 42] are reported next.

THEOREM 3.35 *For a program P and $Post \subseteq B_L$, the function $\vartheta_{P,Post}$ is monotonic and downward continuous over the lattice $(2^{B_L}, \subseteq)$. Moreover,*
(i) $\vdash \{Pre\}\ P\ \{Post\}$ *iff* $Pre \subseteq \vartheta_{P,Post}(Pre)$,
(ii) $wlp(P, Post) = gfp(\vartheta_{P,Post}) = \vartheta_{P,Post} \downarrow \omega$,
(iii) $wp(P, Post) = \vartheta_{P,Post} \uparrow \omega$,
(iv) $\vdash_t \{\vartheta_{P,Post} \uparrow (\omega)\}\ P\ \{Post\}$ *hold by the level mapping*

$$|A| = min\{\ i \mid A \in \vartheta_{P,Post} \uparrow i\ \}$$

if $A \in \vartheta_{P,Post} \uparrow \omega$ *and* $|A| = 0$ *otherwise.* □

Intuitively, in *(i)* a characterization of relation $\vdash$ is given, based on pre fixpoints of $\vartheta_{P,Post}$. In particular, in *(ii)* we state that the greatest fixpoint (which coincides with the downward ordinal closure) is the weakest liberal precondition of P and $Post$. *(iii)* shows that the weakest precondition coincides with the upward ordinal closure of $\vartheta_{P,Post}$ (although, it does not coincide with the least fixpoint of $\vartheta_{P,Post}$.) Finally, *(iii-iv)* is an alternative proof of Theorem 3.16, by defining the level mapping $|\ |$ starting from $\vartheta_{P,Post}$.

3.9 Correct and Computed Instances

So far, we stressed the fact that postconditions characterize correct instances of the intended queries, as formally stated by Corollary 3.19 *(ii)*. However, under certain rather general assumptions, the proposed proof method can be also employed to achieve a characterization of the computed instances of queries. In this section, we present two methods for characterizing computed instances, applicable in different situations.

Given a set of atoms I, we define $Min(I) = \{A \in I \mid \neg\exists B \in I : B < A\}$, i.e. the set of minimal elements of I under the instantiation ordering $<$. We recall that Σ_L is the set of function symbols of the underlying language L.

THEOREM 3.36 *Assume that Σ_L contains infinitely many symbols, and $\vdash \{Pre\}\ P\ \{Post\}$, $\vdash \{Pre\}\ Q\ \{Post\}$ hold with $Post$ well-supported.*
Consider the set $\mathcal{Q} = \{Q\theta \mid Post \models Q\theta\}$:

(i) $\mathcal{Q}$ is the set of correct instances of P and Q,

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

(a) $Min(\mathcal{Q}) = \mathcal{Q}$,

(b) Q is a set of ground queries,

(c) Q is a finite set.

Proof. *(i)* Since $Post$ is well-supported, $Post \models Q\theta$ iff $sp(P, Pre) \models Q\theta$ iff $M_P^L \cap Pre \models Q\theta$. If $P \models Q\theta$ then $M_P^L \cap Pre \models Q\theta$, and by the equivalences above, $Q\theta \in \mathcal{Q}$. On the other hand, if $Post \models Q\theta$ then $M_P \models Q\theta$. Since Σ_L is infinite, $M_P \models Q\theta\gamma$, where γ maps every variable of $Q\theta$ into a distinct ground term with functor not appearing in P or $Q\theta$. $M_P \models Q\theta\gamma$ implies $P \models Q\theta\gamma$, and by the Theorem on Constants (see [50]), $P \models Q\theta$.

(ii)(a) Let $\mathcal{Q}'$ be the set of computed instances of P and Q. By Theorem 3.19*(ii)* $\mathcal{Q}' \subseteq \mathcal{Q}$. Conversely, consider $Q' \in \mathcal{Q}$. By *(i)* $P \models Q'$. Therefore, by completeness of LD-resolution there exists $Q'' \in \mathcal{Q}'$ more general than Q'. As pointed out above $Q'' \in \mathcal{Q}$. This and the hypothesis $Min(\mathcal{Q}) = \mathcal{Q}$ imply that Q' is a variant of Q''. As $\mathcal{Q}'$ is closed under the variant-relation, we conclude $Q' \in \mathcal{Q}'$.

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

(ii)(c) Let us show by contrapositive that *(ii)(c)* $\Rightarrow$ *(ii)(b)*. If $Q' \in \mathcal{Q}$ is not ground then $Post \models Q'$ implies $Post \models Q''$ for every ground instance Q'' of Q'. Since Σ_L is infinite, there are infinitely many ground instances of Q', and then $\mathcal{Q}$ is infinite. □

In particular, for atomic queries Q, hypothesis *(ii)(b)* can be rewritten as $\mathcal{Q} \subseteq Post$.

As an example, consider the program `EVEN`

```
even(0).
even(s(s(X)))  ← even(X).
```

(defined on a language with Σ_L infinite) and the query `even(X)` with $Pre = B_L$, and $Post$= $\{\texttt{even(s}^{2k}\texttt{(0))} \mid k \geq 0\}$. It is easy to show that $\vdash \{Pre\}$ `EVEN` $\{Post\}$ holds, and that $Post$ is well-supported. Also, we have that $\mathcal{Q}$ coincide with $Post$, and then it is a set of ground queries. By Theorem 3.36*(ii)(b)*, we have that $\mathcal{Q}$ coincides with the set of computed instances of `EVEN` and `even(X)`.

Recently, Apt et al. [8] 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, which is explained below. Roughly, such programs have the property of delivering non-redundant computed instances of the intended queries. Here, a set $\mathcal{Q}$ of queries is called *subsumption-free* if for no pair Q, Q' of non-variant distinct queries from $\mathcal{Q}$ we have that Q is an instance of Q'.

DEFINITION 3.37 A program P is said to be $(Pre, Post)$*-redundancy-free* if

- $\vdash \{Pre\}\ P\ \{Post\}$, and
- for any query Q such that $\vdash \{Pre\}\ Q\ \{Post\}$, the set of computed instances of Q is subsumption-free. □

Thus, $(Pre, Post)$-redundancy-freedom guarantees that no redundant output is obtained as a result of computing with any intended query. The next result, adapted from [8], shows that, for $(Pre, Post)$-redundancy-free programs which are partially correct w.r.t. $(Pre, Post)$, the computed instances of the intended queries can be retrieved from $Post$.

THEOREM 3.38 *Assume that Σ_L contains infinitely many symbols.*
Consider a $(Pre, Post)$-redundancy-free program P with $Post$ well-supported w.r.t. P and Pre, and a query Q such that $\vdash \{Pre\}\ Q\ \{Post\}$. Then the set of computed instances of Q is $Min\{Q\theta \mid Post \models Q\theta\}$.

Proof. Corollary 7.2*(iii)* from [8] makes the same assumptions. However, in the conclusion it is stated that the computed instances of Q are the most general instances of Q which are true in $M_P^L \cap Pre$. Our conclusion follows by considering that for Q' instance of Q $M_P^L \cap Pre \models Q'$ iff $Post \cap Pre \models Q'$ since $Post$ is well-supported, and $Post \cap Pre \models Q'$ iff $Post \models Q'$ since $\vdash \{Pre\}\ Q\ \{Post\}$. □

Thus for $(Pre, Post)$-redundancy-free programs the computed instances of the intended queries can be retrieved from the well-typed fragment of the least Herbrand model. We are now faced with the problem of proving that a given program is $(Pre, Post)$-redundancy-free. Using Definition 3.37 directly is not handy, as the notion of redundancy-freedom is essentially procedural. To solve this problem, several sufficient conditions are studied in [8] based either on syntactic decidable properties or on declarative semantics, which allow us to prove that a given program is (Pre, $Post$)-redundancy-free.

THEOREM 3.39 *Assume that $\vdash \{Pre\}\ P\ \{Post\}$. If the following conditions hold*

SEM1. *If $A \leftarrow B_1, \ldots, B_n$ and $A \leftarrow C_1, \ldots, C_k$ are ground instances of two different clauses in P then*

$$Post \not\models H \wedge B_1, \ldots, B_n \wedge C_1, \ldots, C_k .$$

SEM2. *If $A \leftarrow B_1, \ldots, B_n$ and $A \leftarrow C_1, \ldots, C_k$ are distinct ground instances of the same clause in P then*

$$Post \not\models H \wedge B_1, \ldots, B_n \wedge C_1, \ldots, C_k .$$

then P is $(Pre, Post)$-redundancy-free.

Proof. By Theorem 3.20, $M_P \cap Pre \subseteq Post$.
Therefore $Post \not\models H \wedge B_1, \ldots, B_n \wedge C_1, \ldots, C_k$ implies

$$M_P \cap Pre \not\models H \wedge B_1, \ldots, B_n \wedge C_1, \ldots, C_k .$$

Therefore [SEM1] and [SEM2] imply respectively the two conditions assumed in [8, Theorem 7.4] to prove that P is $(Pre, Post)$-redundant-free. □

In certain situations the conditions of Theorem 3.39 can be ensured by means of syntactic restrictions. Namely, condition SEM1 is obviously implied by condition

SYN1. If $A_1 \leftarrow B_1, \ldots, B_n$ and $A_2 \leftarrow C_1, \ldots, C_k$ are variable disjoint variants of different clauses in P, then A_1 and A_2 do not unify,

and condition SEM2 is automatically satisfied when the following condition holds:

SYN2. If $A \leftarrow B_1, \ldots, B_n \in P$, then $Var(B_1, \ldots, B_n) \subseteq Var(A)$.

We can now summarize our methodology for characterizing computed instances on the basis of relation $\vdash$.

(1) *Construct Pre and Post such that* $\vdash \{Pre\}\ P\ \{Post\}$ *and* $\vdash \{Pre\}\ Q\ \{Post\}$ *hold and Post is well-supported w.r.t. P and Pre.*

(2) *Prove that P is (pre, post)-redundancy-free using Theorem 3.39.*

(3) *Find a correct instance* Q' *of* Q, i.e. such that $Post \models Q'$.

(4) *By suitably generalizing from* (3) *find a minimal correct instance* Q' *of* Q, i.e. such that $Post \models Q'$ and $Post \not\models Q''$ for every $Q'' < Q'$. (In general, find the set of minimal correct instances of Q).

(5) *Apply Corollary 3.38.*

We refer the reader to [8] for several examples and a comparison of this approach with others related to $\mathcal{S}$-semantics. We will show an example using these results in Section 4.4.

4 Applications

The objective of this section is to show that the proof method, or an equivalent formulation of it, is of practical use in real applications. We proceed by showing how to reason on arithmetic built-in's, preventing the selection of ill-typed arithmetic atoms. Then we introduce a relation $semi{-}\vdash_t$ which, although equivalent to $\vdash_t$, is more suitable for modular program development. Moreover, we provide a sufficient condition, in the form of restrictions to the admissible Pre and $Post$, which ensures the safe omission of the occur check in the unification algorithm used to resolve a query. The occur check is usually omitted in Prolog's implementations. Finally, we briefly overview some already well-studied applications of the proof method to meta-programming, and discuss a complete example.

4.1 Arithmetic Built-in's

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

$$\texttt{<, <=, =:=, =/=, is, >=, >}$$

can appear (in clause bodies only). A formalization of Prolog's interpretation of arithmetic relations within logic programming is due to Kunen [33]. According to [51], 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 ground arithmetic expressions (in short, gae's.) Formally, the set Gae of gae's coincides with the Herbrand universe on the signature

$$\Sigma_{Ar} = \{\texttt{0, 1, 2 ..., +}^2\texttt{, *}^2\texttt{, /}^2\texttt{, mod}^2\}.$$

We denote by $Gaec$ the set of numerals {`0, 1, -1, 2, -2` ...}.

If n, m are gae's then we stipulate that the LD-derivation fails if $value(n)$, i.e. the natural number denoted by n, is equal or greater than $value(m)$, i.e. the natural number denoted by m. If the value of n is smaller than that of m, the resolvent is the rest of the goal. This is the procedural semantics of `<` in Prolog. A similar operational semantics is given for `<=, =/=, >=, >` , whereas for `is` only the second argument is required to be a gae.

Unfortunately, as discussed in [7], 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. According to [33], we extend LD-resolution assuming that a program with arithmetic in which $<$ appear, implicitly contains the set of clauses

$$Post_{<} = \{\ n\ \texttt{<}\ m \mid n, m \in Gae \wedge value(n) < value(m)\ \}$$

(and analogously for `<=, =/=, >=, >` .) If `is` appears in the program, we consider the set

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

Consider now the program `PART`

```
part(X, [Y|Xs], [Y|Ls], Bs ) ← X > Y, part(X, Xs, Ls, Bs).
part(X, [Y|Xs], Ls, [Y|Bs] ) ← X ≤ Y, part(X, Xs, Ls, Bs).
part(X, [], [], [] ).
```

for partitioning a list of gae's, and suppose that $\vdash \{Pre\}$ `PART` $\{Post\}$ holds for some $Pre, Post$. By Corollary 3.19 *(i)*, for every selected atom A

$$Pre \models A. \tag{2}$$

This suggests us a simple conditions for preventing the selection of ill-typed arithmetic atoms, consisting of imposing that if (2) 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

$$Pre_{<} = \{\ n\texttt{<}m \mid n, m \in Gae\ \}$$

then $Pre \models n$ `<` m implies that n, m are gae's, under the weak hypothesis that there exists at least one symbol `f` in Σ_L that does not belong to Σ_{Ar}. In fact, we notice that n `<` m is ground, otherwise by instantiating the variables of n, m with a ground term containing `f` we do not get two gae's, even tough $Pre \models n$ `<` m. Second, since n `<` m is ground, we have that n `<` $m \in Pre$ and then m, m are gae's.

We reason analogously for the other arithmetic atoms, except for `is`, for which we define

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

Considering again `PART`, we define the pre- and postcondition as follows:

$$\begin{aligned} Pre_{\texttt{PART}} &= Pre_{>} \cup Pre_{\leq} \cup \{\ \texttt{part(}x\texttt{, }xs\texttt{, }ls\texttt{, }bs\texttt{)} \mid \\ &\quad x \in Gae \wedge xs \in List(Gae)\} \\ Post_{\texttt{PART}} &= Post_{>} \cup Post_{\leq} \cup \{\ \texttt{part(}x\texttt{, }xs\texttt{, }ls\texttt{, }bs\texttt{)} \mid \\ &\quad x \in Gae \wedge xs, ls, bs \in List(Gae) \wedge ls < x \geq bs\} \end{aligned}$$

By $ls < x \geq bs$ we mean that every element in the list ls (resp., bs) is smaller (resp., greater or equal) than x.

It is readily checked that $\vdash \{Pre_{\texttt{PART}}\}\ \texttt{PART}\ \{Post_{\texttt{PART}}\}$. Therefore, when an arithmetic atom $n < m$ is selected in a LD-derivation for `PART` and a query Q such that $\vdash \{Pre_{\texttt{PART}}\}\ Q\ \{Post_{\texttt{PART}}\}$, we have that

$$Pre_{<} \models n < m. \tag{3}$$

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 Consider a program with arithmetic P (resp., a query with arithmetic Q) with the underlying language L such that $\Sigma_L \setminus \Sigma_{Ar} \neq \emptyset$.
We write $\vdash \{Pre\}\ P\ \{Post\}$ (resp., $\vdash \{Pre\}\ Q\ \{Post\}$), if $\vdash \{Pre\}\ P\ \{Post\}$ (resp., $\vdash \{Pre\}\ Q\ \{Post\}$) holds for P (resp., Q) as a logic program, and for every arithmetic predicate `op`, the sets of `op`-atoms in Pre and in $Post$ coincide with $Pre_{\texttt{op}}$ and $Post_{\texttt{op}}$, respectively. □

We point out that in the case of programs with arithmetic, the resulting LD-trees are still finitely branching, since by the definition of their operational semantics there is at most one resolvent when an arithmetic atom is selected. This observation together with Definition 4.1 allow us to extend the Lifting Lemma, and *a fortiori* the Strong Completeness Theorem for LD-resolution, by means of the following result.

LEMMA 4.2 *Assume that* $\vdash \{Pre\}\ P\ \{Post\}$ *and* $\vdash \{Pre\}\ Q\ \{Post\}$ *hold for a program with arithmetic* P *and a query with arithmetic* Q*. Then no LD-derivation for* P *and* Q *ends in an error.* □

By noting that the LD-derivation obtained applying the Lifting Lemma for logic programs to programs with arithmetic does not end in an error by Lemma 4.2, we can extend the Completeness of LD-resolution, under the same hypothesis of Lemma 4.2.

THEOREM 4.3 (COMPLETENESS OF LD-RESOLUTION)
Assume that $\vdash \{Pre\}\ P\ \{Post\}$ *and* $\vdash \{Pre\}\ Q\ \{Post\}$ *hold for a program with arithmetic* P *and a query with arithmetic* Q*. Then for every correct instance* Q' *of* Q *there exists a computed instance of* Q *more general than* Q'*.* □

By the last two results and Definition 4.1, the proof theory and the related results are generalized to programs with arithmetic.

We conclude this section with a note on definiteness conditions for gae's. According to the operational semantics of other programming languages, we may want to prove absence of other run-time errors such as *division by zero*, as in the example `2 < 3/0`. We can easily achieve that, by removing from $Pre_{<}$ all the atoms containing a division by zero. With this modification, if an atom $n < m$ is selected then $n, m \in Pre_{<}$, and then no division by zero appears in n or m. The same reasoning applies to the other predicates, except for `is`, for which we have to remove the atoms t `is` n such that n contains a division by zero.

4.2 Modular Verification

The definition of relation $\vdash_t$ has a major drawback, due to lack of expressiveness of level mapping in a modular approach to verification. We introduce the problem with an example. Consider the program SUBLIST

```
sublist(Xs, Ys) ← Xs is a sublist of Ys.

sublist(Xs, Ys) ←
   append(_, Zs, Ys),
   append(Xs, _, Zs).
```

augmented by the APPEND program.

A level mapping such that

$$|\texttt{append}(_,_,[x_1,\ldots,x_n])| = n$$

is a natural candidate to show $\vdash_t \{Pre_{\texttt{APPEND}}\}$ APPEND $\{Post_{\texttt{APPEND}}\}$, where

$$Pre_{\texttt{APPEND}} = \{\texttt{append}(_,_,xs) \mid xs \in List(\alpha)\}$$
$$Post_{\texttt{APPEND}} = \{\texttt{append}(xs,\ ys,\ zs) \mid zs = xs * ys\}.$$

Analogously, considering the precondition for SUBLIST:

$$Pre_{\texttt{SL}} = \{\texttt{sublist}(_,xs) \mid xs \in List(\alpha)\} \cup Pre_{\texttt{APPEND}}$$

an intuitively correct level mapping is $|\texttt{sublist}(_,[x_1,\ldots,x_n])| = n$. However, the proof obligations of Definition 3.2 require that

$$|\texttt{sublist}(_,[x_1,\ldots,x_n])| > |\texttt{append}([x_1,\ldots,x_n],_,_)| = n.$$

Therefore, to show $\vdash_t \{Pre_{\texttt{SUBLIST}}\}$ SUBLIST $\{Post_{\texttt{SUBLIST}}\}$, we must consider a somewhat unnatural level mapping such as $|\texttt{sublist}(_,[x_1,\ldots,x_n])| = n+1$. Unfortunately, such a phenomena propagates upward when considering programs which use SUBLIST.

In addition, the counter-intuitiveness of the level mapping is not the only drawback of relation $\vdash_t$. To some extent, one can assume that the implementation of APPEND is hidden, and only its specification is known, which consists of pre- and postconditions plus a level mapping. Suppose now that the implementation of APPEND is modified later on to the following one (which includes type checks):

```
append([], Xs, Xs) ←
   list(Xs).
append([X|Xs], Ys, [X|Zs]) ←
   append(Xs, Ys, Zs).

list([]).
list([X|Xs]) ←
   list(Xs).
```

As in the case for SUBLIST, the level mapping for APPEND has to be set as follows

$$|\texttt{append}(_,_,[x_1,\ldots,x_n])| > n.$$

This fact modifies the specification of APPEND. All of the modules that use APPEND have to be reconsidered, and as a result this problem propagates upward. Thus, the objective of modularity is not achieved.

In the following we provide a relation $semi{-}\vdash_t$ (introduced in [41] following the approach of [4]), which is equivalent to $\vdash_t$, but addressed the modularity problems.

Relation $semi\!-\vdash_t$

First, for two predicate symbols p and q, we write $p \sqsupset q$ if p uses q in its definition, but p and q are not mutually recursive; we write $p \simeq q$ if p and q are mutually recursive. Then, we define the relation $semi\!-\vdash_t$, which will be shown to be equivalent to $\vdash_t$.

DEFINITION 4.4 Given a program P and a specification $(Pre, Post)$, we write $semi\!-\vdash_t \{Pre\}\ P\ \{Post\}$ iff there exists a level mapping $|\ |$ such that for every $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$

1. for $i \in [1, n]$:

$$Pre \models A \ \wedge \ Post \models B_1, \ldots, B_{i-1} \Rightarrow$$

$$\textit{(a)}\ Pre \models B_i \quad \text{and}$$

$$\textit{(b)}\ |A| > |B_i| \ \textit{if}\ rel(A) \simeq rel(B)$$
$$|A| \geq |B_i| \ \textit{if}\ rel(A) \sqsupset rel(B)$$

2. $Pre \models A \ \wedge \ Post \models B_1, \ldots, B_n \Rightarrow Post \models A$ □

Differently from Definition 3.2, in *1(b)* we now distinguish two cases depending on the fact that $rel(A)$ and $rel(B_i)$ are (or not) mutually recursive predicates. If they are mutually recursive, a strict decreasing is imposed.

No change is required for relation $\vdash_t$ for queries. As a first observation, we point out that $\vdash_t \{Pre\}\ P\ \{Post\}$ implies $semi\!-\vdash_t \{Pre\}\ P\ \{Post\}$. Indeed, the two relations do coincide, as the following theorem shows. The proof of the Theorem as well as several other results on relation $semi\!-\vdash_t$ can be found in [41].

THEOREM 4.5 *For a program P and a specification $(Pre, Post)$:*

$$\vdash_t \{Pre\}\ P\ \{Post\}\ \textit{holds} \quad \textit{iff} \quad semi\!-\vdash_t \{Pre\}\ P\ \{Post\}\ \textit{holds.}$$ □

This result allows us to extend all of the properties of triples in relation $\vdash_t$ to triples in $semi\!-\vdash_t$.

Proof Outlines

Proof outlines for $semi\!-\vdash_t$ are a slight variant of those for $\vdash_t$. The only change is in the proof obligation *(v)*, that becomes

(v′) for $i \in [1, n]$

$$g'_0 \wedge f'_1 \wedge \ldots \wedge f'_{i-1} \Rightarrow g'_i \wedge \begin{cases} t'_0 > t'_i & \text{if } rel(A_0) \simeq rel(A_i) \\ t'_0 \geq t'_i & \text{if } rel(A_0) \sqsupset rel(A_i) \end{cases}$$

Consider `SUBLIST` again. By defining

$$|\texttt{append}(_, _, [x_1, \ldots, x_n])| = n$$
$$|\texttt{sublist}(_, [x_1, \ldots, x_n])| = n$$

it results that $semi\!-\vdash_t \{Pre_{\texttt{SL}}\}\ \texttt{SUBLIST}\ \{Post_{\texttt{SL}}\}$ where

$$Post_{\texttt{SL}} = \{\texttt{sublist(}xs\texttt{, }ys\texttt{)} \mid xs \text{ is a sublist of } ys\} \cup Post_{\texttt{APPEND}}$$

The proof outline for the first clause is

$\{\ ys \in List(\alpha)\ \}$
`sublist(Xs, Ys)` $\leftarrow$ $\{\ |Ys|\ \}$
$\quad\{\ ys \in List(\alpha)\ \}$
$\quad$`append(_, Zs, Ys)` $\{\ |Ys|\ \}$
$\quad\quad\{\ ys = _ * zs\ \}$
$\quad\quad\{\ zs \in List(\alpha)\ \}$
$\quad$`append(Xs, _, Zs)` $\{\ |Zs|\ \}$
$\quad\quad\{\ zs = xs * _\ \}$
$\{\ ys = _ * xs * _\ \}$

We point out now that changing the implementation of `APPEND` as we did at the beginning of the Section, does not involve reconsidering the correctness proofs of the modules using `APPEND`, since its specification has not to be modified.

Modular Proofs

The wide applicability of relation $semi\text{-}\vdash_t$ is supported by several results on modular program verification. In [41] several results are shown that allow us to prove that a triple $\{Pre\}\ P \cup P'\ \{Post\}$ is in a relation $\vdash, \vdash_t$ or $semi\text{-}\vdash_t$ starting from proofs that triples for P and P' are in the same relation.

As an example, we report a result on relation $\vdash$. We say that P extends P' if no atom appearing in P' is defined in P. In addition, with $Pre|_\Pi$ we denote the set of $A \in Pre$ such that $rel(A)$ is in Π.

THEOREM 4.6 *Let P, P' be two programs such that P extends P'. Assume that:*
(i) $\vdash \{Pre_P\}\ P\ \{Post_P\}$ *and* $\vdash \{Pre_{P'}\}\ P'\ \{Post_{P'}\}$, *and* $Pre_{P'} = Pre_{P'}|_{\Pi_{P'}}$
(ii) $Pre_P|_{\Pi_{P'}} \subseteq Pre_{P'}$
(iii) $Post_P|_{\Pi_{P'}} = Post_{P'}|_{\Pi_P}$
Then the triple $\vdash \{Pre_P \cup Pre_{P'}\}\ P \cup P'\ \{Post_P \cup Post_{P'}\}$ *holds.*
Moreover, if $Post_{P'}$ is well-supported w.r.t. P' and $Pre_{P'}$, and $Post_P$ is well-supported w.r.t. P and $Pre_P|_{\Pi_P \setminus \Pi_{P'}}$, then $Post_P \cup Post_{P'}$ is well-supported w.r.t. $P \cup P'$ and $Pre_P \cup Pre_{P'}$. □

Intuitively, $Pre_{P'} = Pre_{P'}|_{\Pi_{P'}}$ means that $Pre_{P'}$ contains only atoms with relation symbol appearing in P'. *(ii)* means that the calls in P to predicates defined in P' are correct with the intended precondition $Pre_{P'}$. *(iii)* means that the intended meaning of predicates shared between P and P' meets.

Theorem 4.6 is easily applicable to modular program development. Consider, in fact, a stratification $P_1, \ldots, P_n$ for a program P, i.e. a partition $P_1 \uplus \ldots \uplus P_n$ of P such that for $i \in [2, n]$ P_i extends P_{i-1}. The concept of stratification clearly characterizes modular development, including bottom-up development (design P_i using P_j for $j < i$) and top-down refinement (design P_i in order to refine predicates used in P_j for $j > i$). To prove correctness of P we can apply $n - 1$ times Theorem 4.6, considering for $i \in [2, n]$ $P_1 \cup \ldots \cup P_{i-1}$ and P_i, in order to prove correctness of $P_1 \cup \ldots \cup P_i$. At the last iteration, we have proved correctness of the whole program P.

4.3 The Occur-check Problem

Apt and Pellegrini [5] present a methodology for the safe omission of the occur-check in the Martelli-Montanari unification algorithm [37]. 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 [5] show that for many practical

programs, the occur-check omission is safe, by providing some sufficient conditions. First, we introduce some basic definitions.

DEFINITION 4.7

- Consider a n-ary predicate symbol p. A *mode* for p is a function d_p from $\{1, \ldots, n\}$ in $\{+, -\}$. 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 *input*-linear (resp., *output*-linear) if the family of terms which occur in its input positions (resp., output) is linear, i.e. no variable occurs twice in the family.
- An atom is called *input–output*-disjoint if the family of terms which occur in its input position is variable disjoint with the family of terms which occur in its output-positions.
- A couple 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. □

One sufficient condition to show that a couple of atoms is NSTO provided by Apt and Pellegrini is the following result.

LEMMA 4.8 *Consider two atoms A, B such that*
(i) they are variable disjoint,
(ii) one of them is input–output disjoint,
(iii) one of them is input linear and the other one is output linear.
Then the couple (A, B) is NSTO. □

We use this result together with our proof theory in order to give a sufficient condition for a safe omission of the occur-check along a LD-derivation.

THEOREM 4.9 *Assume that $\vdash \{Pre\}\ P\ \{Post\}$ and $\vdash \{Pre\}\ Q\ \{Post\}$ hold. Consider a set $\Pi \subseteq \Pi_L$ of predicate symbols such that*
(i) for every atom A such that $rel(A) \in \Pi$, if $Pre \models A$ 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 Π is output linear.
Then in every LD-derivation for P and Q the omission of occur-check for all selected Π-atoms.

Proof. Let A be an atom selected in a LD-derivation whose predicate symbol is in Π, and B the head of a variant of a clause from P. We only have to show the hypothesis of Lemma 4.8.

(i) A and B are variable disjoint by *standardization apart*,

(ii) By Corollary 3.19 *(i)*

$$Pre \models A$$

and by *(i)* A is input-linear and input-output disjoint.

(iii) By *(ii)*, B is output linear (since a variant of B is so.)

Therefore, we conclude that (A, B) is NSTO. □

Theorem 4.9 generalizes the results of [5], where a similar result is given for well-moded programs, which are properly included in the class of programs such that $\vdash \{Pre\}\ P\ \{Post\}$. A dual version of the Theorem is obtained by considering sets of clauses instead of sets of predicate symbols.

As an example, consider the program `QUICKSORT`.

```
qs(Xs, Ys) ← Ys is an ordered permutation of the list Xs.

qs([X | Xs], Ys) ←
   part(X, Xs, Littles, Bigs),
   qs(Littles, Ls),
   qs(Bigs, Bs),
   append(Ls, [X | Bs], Ys).
qs([], []).
```

augmented by the `PART` and `APPEND` programs.

It is readily checked that $\vdash \{Pre\}$ `QUICKSORT` $\{Post\}$, where

$$\begin{aligned} Pre \quad &= \quad \{\texttt{qs}(xs,ys) \mid xs, ys \in List(Gae)\ \} \\ &\cup \quad Pre_{\texttt{PART}} \\ &\cup \quad \{\texttt{append}(xs,\ ys,\ zs) \mid xs, ys \in List(Gae)\ \} \\ Post \quad &= \quad \{\texttt{qs}(xs,ys) \mid ys \text{is an ordered permutation of the list } xs\ \} \\ &\cup \quad Post_{\texttt{PART}} \\ &\cup \quad \{\texttt{append}(xs,\ ys,\ zs) \mid zs = xs * ys\ \} \end{aligned}$$

Consider now $\Pi = \{$ `qs`, `part`, `append` $\}$ and the moding

$$\begin{gathered} \texttt{qs}(+,-) \\ \texttt{part}(+,+,-,-) \\ \texttt{append}(+,+,-) \end{gathered}$$

It is readily checked that we are in the hypothesis of Theorem 4.9: the heads of the clauses with predicate symbol in Π are *output* linear since in their output positions no variable appears more than once. Moreover, if $\texttt{qs}(Xs,Ys)$ (resp., $\texttt{part}(X,Xs,Ls,Bs)$ or $\texttt{append}(Xs,\ Ys,\ Zs)$) is true in the precondition then Xs (resp., X and Xs, Xs and Ys) is ground. Therefore, the occur-check can be safely omitted along any LD-derivation for `QUICKSORT`.

4.4 A Complete Example

Consider the following program `DICTIONARY` for retrieving a pair $(key, value)$ in a dictionary organized as a binary search tree (in short, a bst):

```
lookup(X, V, tree((Y,V), L, R)) ← X =:= Y.
lookup(X, V, tree((Y,_), L, R)) ← X < Y, lookup(X, V, L).
lookup(X, V, tree((Y,_), L, R)) ← X > Y, lookup(X, V, R).
```

This program is a simplified version of program 15.9 from Sterling and Shapiro [51]. Here, a bst is represented by either the constant `void`, denoting the empty bst, or by the term $\texttt{tree}((x,\ V),\ L,\ R)$, where x is a gae, V is a term, L and R are bst's, x is greater than the keys occurring in the left subtree, and smaller than the keys occurring in the right subtree. The program uses the arithmetic equality built-in `=:=` (see Section 4.1), which, similarly to `>`, `<`, etc., evaluates both arguments before comparison.

This program has been designed to be queried with bst's in the third argument of `lookup`. As a result the construction of $M^L_{\texttt{DICTIONARY}}$ is particularly awkward. A

ground query `lookup(`x`,`v`,`t`)` with an *unordered* binary tree t, can either succeed or not, depending on the distribution of the keys in the tree. Take now

$$\begin{aligned} Pre &= \{\ \texttt{lookup}(x,v,b) \mid x \text{ is a gae, and } b \text{ is a bst}\ \} \\ &\quad \cup\ Pre_{<} \cup Pre_{>} \cup Pre_{=:=} \\ Post &= \{\ \texttt{lookup}(x,v,b) \mid (x,v) \text{ is in the bst } b\ \} \\ &\quad \cup\ Post_{<} \cup Post_{>} \cup Post_{=:=} \end{aligned}$$

It is easy to see that $\vdash \{Pre\}$ `DICTIONARY` $\{Post\}$ holds, and that $Post$ is well-supported w.r.t. `DICTIONARY` and Pre.

As a first consequence, by Lemma 4.2 we have that no LD-derivation ends in an error when an arithmetic atom is selected.

In addition, consider $Q =$ `lookup(`x`, X, `B`)`, where x is a gae and B is a bst. We calculate the set $Min\{Q\theta \mid Post \models Q\theta\}$:

$$\mathcal{Q} = Variant(\{\ \texttt{lookup}(x,V,B) \mid (x,V) \text{ is in the bst } B\ \}) \tag{4}$$

where, in general, $Variant(I)$ is the set of variants of the atoms I. We will show that $\mathcal{Q}$ is the set of computed instance of `DICTIONARY` and Q, under the hypothesis that $\Sigma_L \setminus \Sigma_{Ar}$ contains infinitely many symbols, by means of Theorem 3.38.

The hypothesis of this Theorem require that `DICTIONARY` is (*pre*, *post*)-redundancy free: we show that by checking conditions SEM1 and SYN2 of Section 3.9. SYN2 readily applies to the `DICTIONARY` program. To check condition SEM1 of Theorem 3.39, it suffices to consider three ground instances of the three clauses of the program with a common head, namely

```
lookup(x, v, tree((y,v),l,r))  ←  x =:= y.
lookup(x, v, tree((y,v),l,r))  ←  x < y, lookup(x,v,l).
lookup(x, v, tree((y,v),l,r))  ←  x > y, lookup(x,v,r).
```

and observe that, for any two gae x and y, exactly one among `x =:= y`, `x < y` and `x > y` holds in $Post$. This implies that SEM1 applies to the `DICTIONARY` program, which is therefore (*pre*, *post*)-redundancy-free. As a conclusion, following the procedure explained in Section 3.9 we have that the set of computed instances of Q is $\mathcal{Q}$.

Finally, we conclude the analysis of `DICTIONARY` by considering a sufficient case for the safe omission of the occur check. Consider $\Pi =$ { `lookup` } and the moding

$$\texttt{lookup}(+,-,+)$$

Condition *(ii)* of Theorem 4.9 is trivially satisfied. To satisfy *(i)*, we need to make a further assumption, namely that the elements in the bst's that we consider belong to a finite set $\alpha \subseteq B_L$. In this case, if $Pre \models$ `lookup(`X`, `V`, `T`)` then

- X is a gae, and then it is ground,
- and T is a bst with elements Y such that every ground instance of Y is in α. As α is finite, every Y is in α and then it is ground. Therefore T itself is ground, and then *(i)* is satisfied.

4.5 Meta-interpreters and Meta-variables

Meta-circular interpreters have been introduced as a fundamental feature of advanced programming languages. In the computational logic paradigm, meta-programming is

a natural and powerful tool. Since the early studies (see [17]), it has become clear the broad scope of this research area, whose applications now include expert systems, semantics and implementation, prototyping and debugging, among others.

A number of meta-interpreters have been introduced 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 is interested in verifying. No uniform and general method has been proposed for tackling the problem by simple and powerful tools. In addition, the proofs of correctness are typically based on *operational reasonings*, without any chance to exploit the declarative reading of logic programs.

The main goal of the paper [43] is to introduce a general criterion for reasoning about meta-interpreters, and demonstrating its applicability. The basic idea of applying the general purpose verification method based on relations $\vdash$ and $\vdash_t$ to the case study of the `Vanilla` meta-interpreter leads to several interesting results when applied to the general case, by relating the pre- and postconditions of the object program to those of the meta-program.

We found that many known properties of meta-interpreters can be proved in a direct way using the proposed method, often in a generalized sense. The contribution of the cited paper can be summarized as follows: under certain natural assumptions, all interesting verification properties lift up from the object program to the meta-program, including

- partial correctness,
- termination,
- absence of errors,
- call patterns persistence,
- correct instances of queries,
- computed instances of queries.

Interestingly, it is possible to establish these results on the basis of purely declarative reasoning, using the proof theory developed in this paper. New results are obtained, including results related to amalgamation, reflection down/up, parameterization and extensions, termination, absence of errors, and characterization of correct and computed instances of queries. We refer the reader to [43] for further details.

In the rest of this section, we concentrate on a related topic, namely Prolog's meta-variables. Meta-variables are used, as an example, to implement reflection down, such as in the implementation of the Prolog meta-predicate `call` [51]:

```
call( X ) ← X.
```

Strictly speaking, this is not a Horn clause, as a variable appears in the body in an atom position. Besides, the use of meta-variables may lead to run time errors: a computation ends in an error when an uninstantiated meta-variable is selected.

To properly tackle the problem we assume the use of the *terms-as-atoms* ambivalent syntax [32]. In our context of ambivalent syntax, we simply assume $\Sigma_L = \Pi_L$ (and then $U_L = B_L$), and provide a sufficient condition for avoiding that LD-derivations end in an error.

First of all, we note that the definition of $\vdash$ applies also to programs which uses meta-variables. In fact, Definition 3.2 makes sense even if arbitrary instances of clauses with meta-variables are considered: pre and postconditions can be used to describe the meaningful instances of the clause.

THEOREM 4.10 *Assume that* $\vdash \{Pre\}\ P\ \{Post\}$ *and* $\vdash \{Pre\}\ Q\ \{Post\}$ *hold on a language* L *such that* $\Sigma_L = \Pi_L$. *If* $Pre \neq B_L$ *then no meta-variable is selected in a LD-derivation for* P *and* Q.

Proof. Suppose, by contradiction, that a meta-variable is selected in a LD-derivation for P and Q, and let X, Q' be the first query in the derivation where a meta-variable X is selected. By the Persistency Theorem 3.18, $\vdash \{Pre\}\ X, Q'\ \{Post\}$ holds. In particular, for every ground instance t, Q'' of X, Q', we have that $Pre \models t$. This implies that $U_L = B_L = Pre$, which is a contradiction. □

4.6 Language Extension

So far, we have assumed that L is the fixed underlying language of the programs under consideration. This assumption is very useful in many cases, such as when considering modular proofs of correctness or when posing a query which contains symbols not appearing in the original program. For instance, the alternative solution of assuming L_P, the language generated by P, does not permit to reuse the proofs of correctness in a modular environment, because the underlying language changes from L_P to $L_{P \cup Q}$.

In this section, we report some results showing that the relations $\vdash$ and $\vdash_t$ are essentially language independent. More precisely, it is immediate to observe that if $\vdash \{Pre\}\ P\ \{Post\}$ holds in some language then, by considering $Pre \cap B_P$ and $Post \cap B_P$, it results that $\vdash \{Pre \cap B_P\}\ P\ \{Post \cap B_P\}$ holds considering L_P as the underlying language. In addition, the next Theorem shows that the relation $\vdash$ can be conservatively extended from L_P to any other language extending L_P.

THEOREM 4.11 *Consider* $\vdash \{Pre\}\ P\ \{Post\}$ *defined on* L. *Let* L' *be a language extending* L. *Then* $\vdash \{Pre^u\}\ P\ \{Post^u\}$ *where*

$$Pre^u = \bigcup_{B \in Atom_L \ \wedge\ Pre \models B} [B]_{L'} \qquad Post^u = \bigcup_{B \in Atom_L \ \wedge\ Post \models B} [B]_{L'}.$$

Proof. Consider a clause $C = A \leftarrow B_1, \ldots, B_n$ from P, and a ground clause $C\theta \in ground_{L'}(P)$. Consider for every $x/t \in \theta$ the transformation x/t' where t' is t where every maximal subterm with the external functor in $f \in \Sigma_{L'} \setminus \Sigma_L$ is replaced by a variable x_f depending on f, and fix $\theta' = \{x/t' \mid x/t \in \theta\}$.

If $Pre^u \models A\theta$ then there exists some atom B in L such that $Pre \models B$ and $A\theta \in [B]_{L'}$. Since $A\theta'$ is the least atom in the $\leq$ such that $A\theta$ is an instance of it and it is an atom in L, we have that $A\theta'$ is an instance of B, and then $Pre \models A\theta'$. Analogously, one can show that if $Post^u \models A\theta$ then $Post \models A\theta'$. In addition, we observe that $C\theta'$ is an instance of C on the language L.

For $i \in [1, n]$, if

$$Pre^u \models A\theta \wedge Post^u \models (B_1, \ldots, B_{i-1})\theta. \tag{5}$$

then, as seen above, $Pre \models A\theta' \wedge Post \models (B_1, \ldots, B_{i-1})\theta'$. Since $\vdash \{Pre\}\ P\ \{Post\}$ and $C\theta'$ is an instance of C on the language L, we conclude that $Pre \models B_i\theta'$. By definition of Pre^u, we have $Pre^u \models B_i\theta$. Analogously, one shows that Definition 3.2 *(2)* holds. □

As a consequence, we have that, for $A \in Atom_L$

$$Pre \models A \text{ iff } Pre^u \models A \quad \text{and} \quad Post \models A \text{ iff } Post^u \models A$$

and then, for a query Q defined in L

$$\vdash \{Pre\}\ Q\ \{Post\} \quad \text{iff} \quad \vdash \{Pre^u\}\ Q\ \{Post^u\}$$

An analogous result holds for $\vdash_t$ by considering the level mapping $|\ |' : B_L \rightarrow N$ such that $|A\theta| = |A\theta''|$, where θ'' is obtained from θ by replacing every x/t with x/t'' where t'' is t where every maximal subterm with the external functor in $f \in \Sigma_{L'} \setminus \Sigma_L$ is replaced by a ground term of L, t_f depending on f.

Consider as a simple example, the set of ground `append`-atoms whose first argument is a list on the language L. We can represent this set in the suggestive form

$$\texttt{append}(list_L \times U_L \times U_L)$$

Consider now $\texttt{append}(list_L \times U_L \times U_L)^u$. The operator ^{u} is intuitively extensible to terms, instead of atoms, and by its definition we have that

$$\texttt{append}(list_L \times U_L \times U_L)^u = \texttt{append}(list_L^u \times U_L^u \times U_L^u)$$

Again, by definition of ^{u}, we observe that $U_L^u = U_{L'}$ and $list_L^u = list_{L'}$. In conclusion

$$\texttt{append}(list_L \times U_L \times U_L)^u = \texttt{append}(list_{L'} \times U_{L'} \times U_{L'}).$$

A result closely related to Theorem 4.11 is expressed below. We are now interested in characterizing the pre- and postconditions on L' by means of a homomorphism $\mathbf{H}$.

THEOREM 4.12 *Assume that* $\vdash_t \{Pre\}\ P\ \{Post\}$ *and* $\vdash_t \{Pre\}\ Q\ \{Post\}$ *hold by the same level mapping* $|\ |$ *w.r.t.* L, *and let* L' *be an extension of* L.
Then $\vdash_t \{Pre'\}\ P\ \{Post'\}$ *and* $\vdash_t \{Pre'\}\ Q\ \{Post'\}$ *w.r.t.* L', *where*

$$Pre' = \mathbf{H}^{-1}(Pre) \qquad\qquad Post' = \mathbf{H}^{-1}(Post)$$

where, $\mathbf{H}(p(t_1, \ldots, t_n)) = p(H(t_1), \ldots, H(t_n))$, *and*

$$H(f(t_1, \ldots, t_n)) = \begin{cases} f(H(t_1), \ldots, H(t_n)) & \text{if } f \in \Sigma_L \\ t_f & \text{if } f \in \Sigma_{L'} \setminus \Sigma_L \end{cases}$$

with t_f, *for every* $f \in \Sigma_{L'} \setminus \Sigma_L$, *is arbitrarily chosen in* U_L.

Proof. Assume that the triples for the relation $\vdash_t$ are proven by means of $|\ |$. We define its extension on L' as follows

$$|A|' = \begin{cases} |\mathbf{H}(A)| & \text{if } rel(A) \in \Pi_L \\ 0 & \text{otherwise} \end{cases}$$

First of all, we need the following simple fact

FACT 1 *Let* A *be an atom defined in* L *and* θ *a ground substitution such that* $A\theta \in B_{L'}$. *Then*
(i) $\mathbf{H}(A\theta) = A\theta^H$ *where* $\theta^H = \{x/H(u) \mid x/u \in \theta\}$,
(ii) $|\ A\theta\ |' = |\ A\theta^H\ |$,
(iii) $Pre' \models A\theta \quad \Leftrightarrow \quad Pre \models A\theta^H \quad \wedge \quad Post' \models A\theta \quad \Leftrightarrow \quad Post \models A\theta^H$.

Proof. *(i)* follows by a straightforward structural induction on the terms of L.
(ii) $|A\theta|' = |\mathbf{H}(A\theta)|$ by definition of $|\ |'$. From *(i)* we have $|\mathbf{H}(A\theta)| = |A\theta^H|$ and then the conclusion.
(iii) By definition $Pre' \models A\theta$ iff $Pre \models \mathbf{H}(A\theta)$. Then the conclusion follows from *(i)*.

□

We show that the triple $\vdash_t \{Pre'\}\ P\ \{Post'\}$ holds. The proofs for $\vdash \{Pre'\}\ P\ \{Post'\}$ and the triples for are very similar. Any ground clause of P is of the form $C\theta = (A \leftarrow B_1, \ldots, B_n)\theta$, where C is a clause from P and θ is a ground substitution. Then for $i \in [1, n]$:

$$
\begin{array}{ll}
 & Pre' \models A\theta \wedge Post' \models (B_1, \ldots, B_{i-1})\theta \\
\Leftrightarrow & \quad \{ \text{ Fact 1}(iii) \} \\
 & Pre \models A\theta^H \wedge Post \models (B_1, \ldots, B_{i-1})\theta^H \\
\Rightarrow & \quad \{ \vdash_t \{Pre\}\ P\ \{Post\}, \text{ and } C\theta^H \in ground_L(P) \ \} \\
 & Pre \models B_i\theta^H \wedge |A\theta^H| > |B_i\theta^H| \\
\Leftrightarrow & \{ \text{ Fact 1}(ii\text{-}iii) \} \\
 & Pre \models B_i\theta \wedge |A\theta|' > |B_i\theta|'.
\end{array}
$$

and, analogously one shows that Definition 3.2 *(2)* holds. □

Analogously to Theorem 4.11, we can use this result in lifting pre- and postconditions up to a richer language.

In the following, we present an example consisting of applying Theorem 4.12 to a special class of level mappings, namely compositional level mappings. Consider a language L and an extension L' of L. A level mapping $|\ |$ on L is called *compositional* iff for every symbol h^n in $\Pi_L \cup \Sigma_L$ there exists a function $\tilde{h} : N^k \rightarrow N$ such that, for every $p^n(t_1, \ldots, t_n) \in B_L$:

$$|p(t_1, \ldots, t_n)| = \tilde{p}(\|t_1\|, \ldots, \|t_n\|)$$

where, for every $f^k(t_1, \ldots, t_k) \in U_L$

$$\|f(t_1, \ldots, t_k\| = \tilde{f}(\|t_1\|, \ldots, \|t_k\|).$$

Fixing a ground term t_f for every $f \in \Sigma_{L'} \setminus \Sigma_L$, we are in the position to apply Theorem 4.12. We have that, for $p^n(t_1, \ldots, t_n) \in B_{L'}$

$$|p(t_1, \ldots, t_n)|' = \begin{cases} \tilde{p}(\|t_1\|', \ldots, \|t_n\|') & \text{if } p \in \Pi_L \\ 0 & \text{otherwise} \end{cases}$$

where, for $f^k(t_1, \ldots, t_k) \in U_{L'}$

$$\|f(t_1, \ldots, t_k)\|' = \begin{cases} \tilde{f}(\|t_1\|', \ldots, \|t_k\|') & \text{if } f \in \Sigma_L \\ \|t_f\| & \text{otherwise} \end{cases}$$

Finally, we observe that $|\ |'$ is still compositional.

As an instance of this example, consider the *list-length* compositional function, defined on ground terms of L as follows:

$$
\begin{array}{l}
\|[x \mid xs]\| = \|xs\| + 1 \\
\|f(t_1, \ldots, t_n)\| = 0 \quad \text{for any } f \in \Sigma_L \ f \neq [\,.\mid .\,].
\end{array}
$$

When lifting to an extended language L', by fixing $t_f = []$ for every new function symbol f, we obtain

$$
\begin{array}{l}
\|[x \mid xs]\|' = \|xs\|' + 1 \\
\|f(t_1, \ldots, t_n)\|' = 0 \quad \text{for any } f \in \Sigma_{L'} \ f \neq [\,.\mid .\,].
\end{array}
$$

4.7 Decidability Issues

In this section we investigate the decidability of the semantics of a logic program. We will show that if $\vdash_t \{Pre\}\ P\ \{Post\}$ holds and $Pre \models \exists A$ for an atom A, then it is decidable whether A belongs to the $\mathcal{C}$ or $\mathcal{S}$-semantics of P.

From Theorem 3.26, we know that if $\vdash_t \{Pre\}\ P\ \{Post\}$ holds then the LD-tree for P and a ground query is finite. Therefore, for every $A \in Pre$, it is decidable whether $A \in \mathcal{M}(P)$. The goal of this section is to present a decision procedure for the $\mathcal{C}$ and $\mathcal{S}$ semantics. First, we point out that, for a (not necessarily ground) atom A, $Pre \models \exists A$ iff there exists a ground instance A' of A such that $\vdash_t \{Pre\}\ A'\ \{Post\}$.

THEOREM 4.13 *Assume that* $\vdash_t \{Pre\}\ P\ \{Post\}$ *holds, and consider an atom* A *such that* $Pre \models \exists A$. *Then*
(i) it is decidable whether $A \in \mathcal{C}(P)$;
(ii) it is decidable whether $A \in \mathcal{S}(P)$.

Proof. Let $x_1, \ldots, x_n$ be the variables of A and $\gamma = \{x_i/t_i \mid i \in [1,n]\}$ such that $A\gamma \in Pre$. By Theorem 4.12 a triple $\vdash \{Pre'\}\ P\ \{Post'\}$ holds w.r.t. the language L' obtained by adding n new constants to L, say $a_1, \ldots, a_n$. We assume that $Pre', Post'$ are defined starting from $t_{a_i} = t_i$, with $t_i \in U_L$. Finally, by defining:

$$\theta = \{x_i/a_i \mid i \in [1,n]\}$$

we have that $A\theta \in Pre'$, and then the LD-tree for P and $A\theta$ is finite.

(i) We point out that

$$\begin{array}{ll} & M_P^{L'} \models_{L'} A\theta \\ \Leftrightarrow & \quad \{A\theta \text{ ground } \} \\ & P \models_{L'} A\theta \\ \Leftrightarrow & \quad \{ \text{ Theorem on constants (see [50]) } \} \\ & P \models_L A \\ \Leftrightarrow & \quad \{ \text{ Definition 2.1 } \} \\ & A \in \mathcal{C}(P). \end{array}$$

Therefore, the decision procedure for $A\theta \in M_P^{L'}$ is also a decision procedure for $P \models_L A$.

(ii) Since the LD-tree for $A\theta$ is finite, we can compute the finite (modulo renaming) set $\{\xi_1, \ldots, \xi_m\}$, $m \geq 0$, of LD-refutations for it and P. By the Lifting Lemma, there exists a computed instance A_i for P and a pure atom B with relation symbol $rel(A)$ by means of a LD-refutation using the same input clauses of ξ_i. Let us show the following fact

$$A \text{ computed instance of } B \text{ and } P \quad \text{iff} \quad A \text{ renaming of } A_i \text{ for some } i \in [1,n] \quad (6)$$

The *if* part is trivial. To prove the *only–if* part, we observe that, if A is a computed instance of B and P with a LD-refutation ξ, then A is a computed instance of A and P with a LD-refutation ξ', which uses the same sequence of clauses of ξ. The conclusion then follows from the observation that $A\theta$ has a LD-refutation ξ'' in P, which uses the same sequence of clauses of ξ'.

We point out that the A_i's are computable since we can compute $\xi_1, \ldots, \xi_m$. Finally, as it is decidable whether two queries are variants, it follows from (6) that it is decidable whether A is a computed instance of B and P, i.e. whether $A \in \mathcal{S}(P)$. □

Theorem 4.13 is adapted from [46], where the particular case $Pre = B_L$ (i.e., acceptable programs) is considered. In addition, in that paper a Prolog implementation of the decision procedure is presented, and the decidability issue is presented as the basis of a tool for testing logic programs. We refer to that paper for further details.

5 General Logic Programs

General programs 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 $comp(P)$ (see [34].) We partly solve this problem by reasoning on the basis of the $\vdash_t$ 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 considering negation only in queries. According to [6], we introduce LDNF$^-$-resolution in the following definition.

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 $\vdash_t \{Pre\}\ Q\ \{Post\}$ for a general query Q iff there exist a level mapping $|\ |$ and $k \in N$ such that for every ground instance $L_1, \ldots, L_n$ of Q
  for $i \in [1, n]$

$$Post \models L_1, \ldots, L_{i-1} \Rightarrow \left\{ \begin{array}{ll} Pre \models A_i \wedge k > |A_i| & \text{if } L_i = A_i \\ Pre \models A_i \wedge k > |A_i| & \text{if } L_i = \neg A_i \end{array} \right\}$$

- 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.

THEOREM 5.2 *Consider a program P and a general query Q such that*

$$\vdash_t \{Pre\}\ P\ \{Post\} \qquad \textit{and } \vdash_t \{Pre\}\ Q\ \{Post\}$$

holds by the same level mapping. Assume that

- *$P \cup \{Q\}$ does not flounder, and*
- *$comp(P) \models Q'$ for Q' instance of Q.*

Then there exists a LDNF$^-$-derivation for P and Q with a computed instance more general than Q'.

Proof. First, we point out that by Theorem 3.21 $\vdash_t \{Pre\}\ P\ \{M_P^L \cap Pre\}$ holds. Let us prove that if $comp(P) \models Q'$ then there exists a LDNF$^-$-refutation for P and Q' with Q' as a computed instance. The proof is by induction on the number n of literals in Q'.

(Base) If Q consists of a literal then we distinguish two cases. If $Q' = A$ then by the Strong Completeness Theorem 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. In addition, by [34, Theorem 16.1] $comp(P) \models \neg A$ implies that there exists a finitely failed SLD-tree for P and A. Since $\vdash_t \{Pre\}\ A\ \{M_P^L \cap Pre\}$, every LD-tree for P and A is finite. Moreover, it is finitely failed. Otherwise, by correctness of LDNF$^-$-resolution $comp(P) \models A$: this is in contradiction with the assumption $comp(P) \models \neg A$ and the fact that $comp(P)$ is consistent.
Thus there exists a LD(NF$^-$) refutation for P and Q'.

(Step) We distinguish two cases.

- If $Q' = A, Q''$ then by the Strong Completeness Theorem there exists a LD(NF$^-$) refutation for P and A. Moreover, by Corollary 3.19*(i)* $M_P^L \cap Pre \models A$ and then $\vdash_t \{Pre\}\ Q''\ \{M_P^L \cap Pre\}$. 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. $comp(P) \models \neg A$ implies that there exists a finitely failed SLD-tree for P and A.
  Since $\vdash_t \{Pre\}\ A\ \{M_P^L \cap Pre\}$, every LD-tree for P and A is finite. Moreover, it is finitely failed. Otherwise, by correctness of LDNF$^-$-resolution $comp(P) \models A$: this is in contradiction with the assumption $comp(P) \models \neg A, Q''$ and the fact that $comp(P)$ is consistent. Thus Q'' is the LDNF$^-$-resolvent of Q'. Moreover, since $A \in Pre \cap FF_P$, Theorem 3.27 and 3.26 imply $M_P^L \cap Pre \models \neg A$, and finally $\vdash_t \{Pre\}\ Q''\ \{M_P^L \cap Pre\}$. The result 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', and thus obtaining the conclusion. □

A completeness theorem is fundamental for reasoning about general programs. Starting from Theorem 5.2, all the results of Section 3 can be extended to general queries. In particular, the following property is crucial in the proof of Theorem 5.2:

$$M_P^L \cap Pre = FF_P^c \cap Pre \tag{7}$$

where FF_P^c is $B_L \setminus FF_P^L$. As we will show in next section, in order to extend Theorem 5.2 to general programs, we assume an additional proof obligation in the definition of relation $\vdash_t$, which is sufficient to extend property (7) to general programs.

5.2 LDNF-resolution

Correctness

The exposition of the approach extended to general programs is organized as follows: first, we extend to general programs the definitions of (weak) total correctness, and of relation $\vdash_t$. 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 5.3 Consider a general program P.

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

(ii) We denote with M_P^L the set of $A \in B_L$ such that there exists a LDNF-refutation for P and A, and with FF_P^L the set of $A \in B_L$ such that there exists a finitely failed LDNF-tree for P and A.

(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 $M_P^L \cap Pre \subseteq Post$ and $Pre \subseteq M_P^L \cup FF_P^L$.

(v) P is *totally correct* w.r.t. a specification $(Pre, Post)$ iff $M_P^L \cap Pre = Post$ and $Pre \subseteq M_P^L \cup FF_P^L$.

□

Although M_P^L and FF_P^L now are not defined declaratively, we will show later that for the class of programs we are interested in they have the expected declarative interpretation. Concerning queries, the definition of relation $\vdash_t$ remains the same of Definition 5.1. Relation $\vdash_t$ is extended to general programs as follows.

DEFINITION 5.4 Let P be a general program, and $(Pre, Post)$ a specification.

- We write $\vdash_t \{Pre\}\ P\ \{Post\}$ iff there exists a level mapping $|\ |$ such that
  (i) for every $A \leftarrow L_1, \ldots, L_n \in ground_L(P)$
  1. for $i \in [1, n]$:

$$Pre \models A \quad \wedge \quad Post \models L_1, \ldots, L_{i-1} \Rightarrow$$

$$\begin{cases} Pre \models B_i \quad \wedge |A| > |B_i| & \text{if } L_i = B_i \\ Pre \models B_i \quad \wedge |A| > |B_i| & \text{if } L_i = \neg B_i \end{cases}$$

  2. $Pre \models A \quad \wedge \quad Post \models L_1, \ldots, L_n \Rightarrow Post \models A$

  (ii) $T_P(Post) \supseteq Post \cap Pre$.
- 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 $\vdash_t$ for general programs is not a conservative extension of that for logic programs. In general, $\vdash_t \{Pre\}\ P\ \{Post\}$ for a logic program P does not necessarily imply that $\vdash_t \{Pre\}\ P\ \{Post\}$ holds for P as a general program. This is due to the additional requirement *(ii)* of Definition 5.4

$$T_P(Post) \supseteq Post \cap Pre.$$

However, we point out that we already met a similar relation: by Theorem 3.32, it is equivalent for logic programs to require that $Post$ is well-supported w.r.t. P and Pre, i.e. $Post \cap Pre = M_P^L \cap Pre$. Therefore, by Theorem 3.28, if $\vdash_t \{Pre\}\ P\ \{Post\}$ holds for a logic program then we can state that $\vdash_t \{Pre\}\ P\ \{M_P^L \cap Pre\}$ holds considering P as a general program. In other words, when dealing with triples for general programs, we are forced to consider *strongest postconditions*.
If condition (ii) of Definition 5.4 is not considered, then we are not able to show the basic properties of correctness, such as the equivalent of Corollary 3.19 for general programs. Consider, as an example the program P:

```
p ← ¬ q.
```

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

In the following theorem, we extend the properties of persistency and correct instance characterization to general programs.

THEOREM 5.5 (PERSISTENCY)
Assume that $\vdash_t \{Pre\}\ P\ \{Post\}$ *and* $\vdash_t \{Pre\}\ Q\ \{Post\}$ *hold by the same level mapping. Then*

(i) for every LDNF-resolvent Q' *of* P *and* Q *the triple* $\vdash_t \{Pre\}\ Q'\ \{Post\}$ *holds by the same level mapping;*

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

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

Proof. *(i)* See Lemma B.2 *(iii)* reported in Appendix B.
(ii) Immediate by *(i)* and Definition 5.1.
(iii) Let $\mathtt{p}^n$ be a fresh predicate symbol, where $x_1, \ldots, x_n$ are the variables of Q. We define Pre' as $Pre \cup \{\ \mathtt{p}^n(t_1, \ldots, t_n) \mid Post \models Q\{x_i/t_i \mid i \in [1,n], t_i \in U_L\}\ \}$, and $Post'$ as $Post \cup \mathtt{p}^n(U_L \times \ldots \times U_L)$. With this assumptions, it is readily checked that $\vdash_t \{Pre'\}\ P \cup \{\ p(x_1, \ldots, x_n) \leftarrow Q\ \}\ \{Post'\}$, and $\vdash_t \{Pre'\}\ Q,\ \mathtt{p}^n(x_1, \ldots, x_n)\ \{Post'\}$, by fixing the level of $p(t_1, \ldots, t_n)$ to the natural number k provided by Definition 5.1. By hypothesis, there exists a LDNF-derivation for P and Q, $\mathtt{p}^n(\mathrm{X}_1, \ldots, \mathrm{X}_n)$ where $\mathtt{p}^n(t_1, \ldots, t_n)$ is selected, with $Q\{x_i/t_i \mid i \in [1,n]\}$ variant of Q'. By *(ii)* and the definition of Pre', we conclude that $Post \models Q\{x_i/t_i \mid i \in [1,n]\}$, and then $Post \models Q'$. □

As a consequence, the Termination Theorem 3.26 extends to general programs.

THEOREM 5.6 (TERMINATION)
Assume that $\vdash_t \{Pre\}\ P\ \{Post\}$ *and* $\vdash_t \{Pre\}\ Q\ \{Post\}$ *hold by the same level mapping. Then every LDNF-tree of* 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 a 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 a 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 NIL

```
p ← ¬ p.
```

and the query p, we have that there exists no LDNF-tree. Therefore, every LDNF-tree is finite. However, the Prolog computation runs forever, by trying to built a subsidiary tree for p, each time a $\neg$ p is selected. In addition, Prolog computation does not check for floundering. We refer the reader to the paper of Apt and Doets [2] for a discussion of the differences between LDNF-resolution and Prolog.

Here, we confine ourselves to observe that with the assumptions of Theorem 5.6 and the additional hypothesis of non-floundering of Q, we are able to state termination w.r.t. the Prolog computation. This is immediate 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 a son, due to the resolution of A with some clause.
Assume now that the Prolog computation runs into an infinite sequence of subsidiary trees. If a ground literal $\neg B$ is selected in a prefix of a LDNF-derivation for A then $|A| > |B|$, where $|\ | : B_L \leftarrow N$ is the level mapping used to show $\vdash_t \{Pre\}\ P\ \{Post\}$ and $\vdash_t \{Pre\}\ Q\ \{Post\}$. Therefore, if there is an infinite sequence $\{A_i\}_{i\geq 0}$ of ground atoms such that A_{i+1} is the root of a subsidiary tree being computed during the resolution of A_i then $|A_{i+1}| > |A_i|$. This is not possible, since N is well-founded.

In particular, $\vdash_t \{Pre\}$ `NIL` $\{Post\}$ does not hold for any $(Pre, Post)$ such that $\mathtt{p} \in Pre$, since we would have to show $|\mathtt{p}| > |\mathtt{p}|$.

Some relevant properties of relation $\vdash_t$ are summarized in the following Lemma, which will be useful later.

LEMMA 5.7 *Assume that* $\vdash_t \{Pre\}\ P\ \{Post\}$ *and* P *is non-floundering. Then*
(i) $Pre \subseteq M_P^L \cup FF_P^L$,
(ii) $M_P^L \cap Pre = FF_P^c \cap Pre$,
(iii) $M_P^L \cap Pre \subseteq Post$,
(iv) $FF_P^L \cap Pre \subseteq Post^c$,
(v) $\vdash_t \{Pre\}\ P\ \{Post \cap Pre\}$.

Proof.
(i) For every $A \in Pre$, by Theorem 5.6 the LDNF-tree for $P \cup \{A\}$ is finite. Since P is non-floundering, then either there exists a LDNF-refutation or it is finitely failed, i.e. $A \in M_P^L \cup FF_P^L$. Therefore, $Pre \subseteq M_P^L \cup FF_P^L$.
(ii). The $\subseteq$ inclusion holds since $M_P^L \subseteq FF_P^c$ by definition of M_P^L and FF_P^c. On the other hand, by *(i)*, $FF_P^c \cap Pre$ is included in M_P^L .
Therefore $FF_P^c \cap Pre \subseteq M_P^L \cap Pre$.
(iii) Consider $A \in M_P^L \cap Pre$. By definition of M_P^L there exists a LDNF-refutation for P and A. By Lemma B.1*(ii)*, $A \in Post$. Therefore, $M_P^L \cap Pre \subseteq Post$.
(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, $FF_P^L \cap Pre \subseteq Post^c$.
(v) Definition 5.4 *(i)* holds by reasoning in the same way of Theorem 3.17. Let us verify Definition 5.4 *(ii)*. We have to show that $T_P(Post \cap Pre) \supseteq Post \cap Pre$. Consider $A \in Post \cap Pre$. By hypothesis, there exists a ground instance of a clause from P

$$A \leftarrow L_1\ , \ldots , L_n$$

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 5.4 *(i)*, $Pre \models B$. Therefore $Post \cap Pre \models L_i$.
- If L_i is a negative literal, say $\neg B$, then $Post \models L_i$ implies $B \notin Post$. Therefore, $B \notin Post \cap Pre$, which implies $Post \cap Pre \models \neg B$. Therefore $Post \cap Pre \models L_i$.

□

We point out that *(ii)* corresponds to property (7), which we already observed to be crucial in the proof of completeness of LDNF$^-$-resolution, namely Theorem 5.2.

A first consequence of the Lemma is the Weak Total Correctness Theorem for general programs.

THEOREM 5.8 (WEAK TOTAL CORRECTNESS) *If* $\vdash_t \{Pre\}\ P\ \{Post\}$ *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 5.7 *(i,iii)*. □

Similarly to positive programs, we can define a notion of strongest postcondition for P and Pre as the intersection of all postconditions $Post$ such that $\vdash_t \{Pre\}\ P\ \{Post\}$. We still denote it by $sp(P, Pre)$. However, in the case of general programs, the notion of strongest postcondition does not result to be a relevant concept. As showed in Theorem 3.32, the condition $T_P(Post) \supseteq Post \cap Pre$ required in Definition 5.4 is equivalent for positive programs to force $Post \cap Pre = M_P^L \cap Pre = sp(P, Pre)$. This fact extends to general programs, as pointed out by the following theorem. Therefore, every postcondition is equivalent to the strongest on the atoms in the precondition.

THEOREM 5.9 *Assume that* $\vdash_t \{Pre\}\ P\ \{Post\}$ *and* P *is non-floundering. Then*
(i) $M_P^L \cap Pre = Post \cap Pre$,
(ii) $FF_P^L \cap Pre = Post^c \cap Pre$,
(iii) $\vdash_t \{Pre\}\ P\ \{M_P^L \cap Pre\}$.

Proof.
(i) The $\subseteq$ inclusion is shown in Lemma 5.7 *(iii)*. On the other hand

$$\begin{array}{ll} & Post \cap Pre \\ & \quad \{ \text{ Lemma 5.7}(i) \text{ and idempotency of } \cap \ \} \\ \subseteq & Post \cap ((M_P^L \cap Pre) \cup (FF_P^L \cap Pre) \\ & \quad \{ \text{ Distributivity } \} \\ = & (Post \cap M_P^L \cap Pre) \cup (Post \cap FF_P^L \cap Pre) \\ & \quad \{ \text{ Lemma 5.7}(iii), (iv) \ \} \\ = & M_P^L \cap Pre \end{array}$$

(ii) This is a direct consequence of the *(i)* and of Lemma 5.7 *(ii,iv)*.
(iii) By Lemma 5.7*(v)*, $\vdash_t \{Pre\}\ P\ \{Post \cap Pre\}$ holds. By *(i)*, $Post \cap Pre$ do coincides with $M_P^L \cap Pre$. Therefore, $\vdash_t \{Pre\}\ P\ \{M_P^L \cap Pre\}$ holds. □

We conclude by stating the Total Correctness Theorem for general programs.

THEOREM 5.10 (TOTAL CORRECTNESS) *If* $\vdash_t \{Pre\}\ P\ \{Post\}$ *holds and* P *is non-floundering, then* P *is totally correct w.r.t. the specification* $(Pre, Post \cap Pre)$.

Proof. By Lemma 5.9*(i, iii)*, $\vdash_t \{Pre\}\ P\ \{M_P^L \cap Pre\}$ holds, with $M_P^L \cap Pre = Post \cap Pre$. By Theorem 5.8, P is weak totally correct w.r.t. the specification $(Pre, Post \cap Pre)$. Noting that $M_P^L \cap Pre = Post \cap Pre$, by Definition 5.3, we conclude that P is totally correct w.r.t. the specification $(Pre, Post \cap Pre)$. □

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

Example: transitive closure

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

```
trans(X, Y, E, V) ←
   member([X, Y], E).

trans(X, Z, E, V) ←
   member([X, Y], E),
   ¬ member(X, V),
   trans(Y, Z, E, [X| V]).

member(X, [X| T]).
member(X, [Y| T]) ←
   member(X, T).
```

Let e be a binary relation on elements in $\alpha \subseteq U_L$ defining a graph. The intended meaning of a query such as `trans(`x`, `y`, `e`, [])` is to check whether $[x, y]$ is in the transitive closure of e. To define suitable pre and postconditions, we write $x \rightsquigarrow_{e \setminus v} y$ 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:

$$\begin{aligned}
Pre = \quad & \{\ \texttt{trans(}x\texttt{, }y\texttt{, }e\texttt{, }v\texttt{)} \mid x \in \alpha \wedge v \text{ list of distinct elements in } \alpha \wedge \\
& \qquad e \in List([\alpha, \alpha]) \wedge \forall\ x \text{ in } v\ \exists\ [x, y] \text{ in } e\ \} \cup \\
& \{\ \texttt{member(}x\texttt{, }ls\texttt{)} \mid x \in \alpha, ls \in List(\alpha)\ \} \cup \\
& \{\ \texttt{member(}x\texttt{, }ls\texttt{)} \mid x \in [\alpha, \alpha], ls \in List([\alpha, \alpha])\ \} \\
Post = \quad & \{\ \texttt{trans(}x\texttt{, }y\texttt{, }e\texttt{, }v\texttt{)} \mid x \rightsquigarrow_{e \setminus v} y\ \} \cup \\
& \{\ \texttt{member(}x\texttt{, }ls\texttt{)} \mid x \text{ is in } ls\ \}
\end{aligned}$$

Next, we define the following level mapping:

$$\begin{aligned}
|\texttt{member(}x\texttt{, }e\texttt{)}| &= |e| \\
|\texttt{trans(}x\texttt{, }y\texttt{, }e\texttt{, }v\texttt{)}| &= 2 \cdot |e| - |v| + 1
\end{aligned}$$

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

$$|\texttt{trans(}x\texttt{, }y\texttt{, }e\texttt{, }v\texttt{)}| \geq 0.$$

In the case $Pre = B_L$, we would need more complicated level mapping (see a similar example in [11]). Here, our precondition simplifies the definition of the level mapping considerably, albeit it is still large enough to reason on the interesting queries. Proving that $\vdash_t \{Pre\}$ `TRANS` $\{Post\}$ holds is easy. As an example, we show a proof obligation relative to the second clause. Consider a ground instance:

```
trans(x, z, e, v) ←
   member([x, y], e),
   ¬ member(x, v),
   trans(y, z, e, [x| v]).
```

such that $Pre \models$ `trans(`x`, `z`, `e`, `v`)` and

$$Post \models \texttt{member([}x\texttt{, }y\texttt{], }e\texttt{), ¬ member(}x\texttt{, }v\texttt{)}.$$

We need to show that:

$$Pre \models \texttt{trans(}y\texttt{, }z\texttt{, }e\texttt{, [}x\texttt{| }v\texttt{])} \quad \text{and} \tag{8}$$

$$|\texttt{trans(}x\texttt{, }z\texttt{, }e\texttt{, }v\texttt{)}| > |\texttt{trans(}y\texttt{, }z\texttt{, }e\texttt{, [}x\texttt{|}v\texttt{])}. \tag{9}$$

For (8), we have that $\exists\ [x, y_1]$ in e is trivially true under the hypothesis that `[`x`, `y`]` is in e. In addition, `[`x `|` v`]` is a list of *distinct* elements in α since so it is v, and x is not in v. Finally y is in α, since `[`x`, `y`]` is in e and e is a list of pairs of elements in α.

For (9), we note that $2 \cdot |e| - |v| + 1 > 2 \cdot |e| - (|v| + 1) + 1$.

The decreasing of the level mapping from the head to the two first body atoms is ensured by noting that $|e| \geq |v|$, and therefore

$$2 \cdot |e| - |v| + 1 > |e|, |v|$$

However, the level mapping may be further simplified to $|$`trans(`x`, `z`, `e`, `v`)`$| = |e| - |v|$, by applying suitable modular proof techniques, such as extensions of those reported in [41] to general programs.

We point out that whenever α is a finite set, any atom $A =$ `member(`X`, `Ls`)` such that $Pre \models A$ is ground. Therefore, by Theorem 5.5 *(ii)*, `TRANS` is non-floundering, since when (an instance) of the negated atom in the second clause is selected then it is ground. Summarizing, by Theorem 5.10, `TRANS` is totally correct w.r.t. the specification $(Pre, Post \cap Pre)$.

Completeness of LDNF-resolution

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

THEOREM 5.11 (COMPLETENESS OF NEGATION AS FAILURE)
Assume that $\vdash_t \{Pre\}\ P\ \{Post\}$ *holds and* $comp(P)$ *is consistent. If for* $A \in Pre$, $P \cup \{A\}$ *does not flounder, then*
(i) if $comp(P) \models \neg A$ *then there exists a finitely failed LDNF-tree for* P *and* A.
(ii) if $comp(P) \models A$ *then there exists a LDNF-refutation for* P *and* A.

Proof. By the Termination Theorem 5.6 every LDNF-tree for P and A is finite. Since $P \cup \{A\}$ is non-floundering either the LDNF-trees are finitely failed or there is a refutation.

(i). The latter case is not possible since by Soundness of LDNF-resolution [34, Theorem 15.6]

$$comp(P) \models A.$$

This is in contradiction with the hypothesis $comp(P) \models \neg A$ and the assumption that $comp(P)$ is consistent.

(ii). The former case is not possible since by Soundness of Negation as Failure [34, Theorem 15.4]

$$comp(P) \models \neg A.$$

This is in contradiction with the hypothesis $comp(P) \models A$ and the assumption that $comp(P)$ is consistent. □

We are now in the position to give a declarative interpretation of M_P^L and FF_P^L.

THEOREM 5.12 *Assume that* $\vdash_t \{Pre\}\ P\ \{Post\}$ *holds,* P *is non-floundering and* $comp(P)$ *is consistent. Then*
(i) $M_P^L \cap Pre = \{\ A \in Pre \mid comp(P) \models A\ \}$ *and*
(ii) $FF_P^L \cap Pre = \{\ A \in Pre \mid comp(P) \models \neg A\ \}$.

Proof. The $\subseteq$ inclusions follow from Soundness of LDNF-resolution and the Negation as Failure Rule. The $\supseteq$ inclusions follow from the Completeness Theorem 5.11. □

The next result extends completeness to general queries.

THEOREM 5.13 (COMPLETENESS OF LDNF-RESOLUTION I)
Assume that $\vdash_t \{Pre\}\ P\ \{Post\}$ *and* $\vdash_t \{Pre\}\ Q\ \{Post\}$ *hold by the same level mapping. Moreover, Assume that* $P \cup \{Q\}$ *does not flounder and* $comp(P)$ *is consistent. If* $comp(P) \models Q$ *then there exists a LDNF-refutation for* P *and* Q*.*

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

Base. If Q consists of only one literal then the result follows by Theorem 5.11.

Step. If $Q = L, Q'$ then by Theorem 5.11 there exists a LDNF-refutation for L. By Lemma 5.7 *(iii,iv)* we have $Post \models L$ and then, by Definition 5.4, $\vdash_t \{Pre\}\ Q'\ \{Post\}$ holds. Therefore we can apply the inductive hypothesis on Q' to have 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., Σ_L infinite), we can state a completeness result that extends a well-know theorem by Cavedon [18].

THEOREM 5.14 (COMPLETENESS OF LDNF-RESOLUTION II)
Assume that $\vdash_t \{Pre\}\ P\ \{Post\}$ *and* $\vdash \{Pre\}\ Q\ \{Post\}$*. Moreover, Assume that* $P \cup \{Q\}$ *does not flounder,* Σ_L *is infinite and* $comp(P)$ *is consistent. If* $comp(P) \models Q'$ *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 $\tilde{Q}'$ 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 t_j, for $j \neq i$. $P \cup \{\tilde{Q}'\}$ 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 5.13, there exists a LDNF-refutation for P and $\tilde{Q}'$. 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 [18] on *acyclic* programs. We recall that a program is *acyclic* if there exists a level mapping $|\ |$ such that

$$\text{for every } A \leftarrow L_1, \ldots, L_n \in ground_{L_P}(P) \text{ for } i \in [1,n] \quad |A| > |L_i|$$

where $|\neg B_i|$ is $|B_i|$. It can be shown that for an acyclic program P, $\vdash_t \{B_L\}\ P\ \{M_P^L\}$ holds in any language L. In addition, $\vdash \{B_L\}\ Q\ \{M_P^L\}$ holds for any query. Therefore, the only hypothesis of Theorem 5.14 are that $P \cup \{Q\}$ does not flounder and that $comp(P)$ is consistent. As a special case of the next result, $comp(P)$ is consistent for acyclic programs P.

THEOREM 5.15 (CONSISTENCY) *The completion of a general program* P *such that* $\vdash_t \{B_L\}\ P\ \{Post\}$ *holds is consistent.*

Proof. In [34] it is shown that I is a model of $comp(P)$ iff $T_P(I) = I$. We show that $T_P(Post) = Post$. The $\subseteq$ inclusion coincides with Definition 5.4*(i-2)*, whereas the other inclusion holds by Definition 5.4 *(ii)*. □

Unfortunately, it is not possible to extend this result to arbitrary preconditions. As an example, $\vdash_t \{\emptyset\}\ P\ \{Post\}$ holds for every P, although not every $comp(P)$ is consistent. The problem is that the completion semantics $comp(P)$ does not take into account pre- and postconditions, and the fact that inconsistency of a subset of $comp(P)$, namely the one not considered in the precondition, causes inconsistency of the entire $comp(P)$. The more intuitive solution to this fact is to put preconditions in the completion, in the same way as we decided to consider $M_P^L \cap Pre$ instead of Pre in the definition of correctness of a logic program. Let us show the idea with an example. Consider the program P

```
p ← ¬ p
q.
```

The inconsistency of $comp(P)$ is uniquely due to the clause for p. If we are interested in a precondition $Pre = \{\ \mathtt{q}\ \}$, then the piece of completion related to Pre is still consistent. Consider the following variant of the completion

$$\begin{aligned} comp(P) \wedge Pre \quad = \quad & EQ \cup \{\ \mathtt{p} \wedge \mathtt{pre_p} \leftrightarrow \mathtt{pre_p} \wedge \neg \mathtt{p}, \\ & \mathtt{q} \wedge \mathtt{pre_q} \leftrightarrow \mathtt{pre_q} \wedge true\ \} \cup \{\ \mathtt{pre_q}, \neg \mathtt{pre_p}\ \} \end{aligned}$$

where EQ is Clark's equality theory. Intuitively, we have inserted in the completion the new predicates $\mathtt{pre_p}$ and $\mathtt{pre_q}$ in order to force consistency on atoms out of Pre. In fact, since $\mathtt{pre_q} \in comp(P) \wedge Pre$, the equivalence

$$\mathtt{q} \wedge \mathtt{pre_q} \leftrightarrow \mathtt{pre_q} \wedge true$$

reduces to

$$\mathtt{q} \leftrightarrow true$$

which is the same as in $comp(P)$. On the other hand the equivalence

$$\mathtt{p} \wedge \mathtt{pre_p} \leftrightarrow \mathtt{pre_p} \wedge \neg \mathtt{p}$$

reduces to $true$ since $comp(P) \wedge Pre \models \neg \mathtt{pre_p}$. On the contrary, the equivalence in $comp(P)$

$$\mathtt{p} \leftrightarrow \neg \mathtt{p}$$

reduces to $false$ and then causes inconsistency.

In conclusion, we argue that $comp(P) \wedge Pre$ is better suited as a semantics for verification than the standard completion semantics, albeit equivalent on the intended queries Q such that $\vdash \{Pre\}\ Q\ \{Post\}$.

6 Conclusions

We mention here two main directions for future research:

- Extension of the verification method to other logic programming paradigms are being investigated, including:
  - (concurrent) constraint programs, on the basis of existing results [22, 38, 47],
  - logic programs with dynamic selection rules, on the basis of existing results in [3, 36],
  - (constraint) deductive databases.
- Design and implementation, on the basis of the verification method, of tools supporting systematic program development. As an example along this direction, the construction of a testing tool is being pursued, on the basis of the results presented in the paper [46].

References

[1] K. R. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[2] K. R. Apt and H. C. Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18(2):177–190, 1994.

[3] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V.S. Alagar and M. Nivat, editors, *Proc. of AMAST'95*, number 936 in Lecture Notes in Computer Science, pages 66–90, Berlin, 1995. Springer-Verlag. Invited Lecture.

[4] K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.

[5] K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 16(3):687–726, 1994.

[6] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 493–574. Elsevier, 1990.

[7] K.R. Apt. *From Logic Programming to Prolog*. CAR Hoare Series Editor. Prentice Hall, 1996.

[8] K.R. Apt, M. Gabbrielli, and D. Pedreschi. A Closer Look at Declarative Interpretations. *Journal of Logic Programming*, 28(2):147–180, 1996.

[9] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6A:743–764, 1994.

[10] K.R. Apt and E.R. Olderog. *Verification of sequential and concurrent programs*. Texts and monographs in computer science. Springer-Verlag, Berlin, 1991.

[11] K.R. Apt and D. Pedreschi. Reasoning about termination of pure prolog programs. *Information and computation*, 106(1):109–157, 1993.

[12] M. Baudinet. Proving Termination Properties of Prolog Programs: a Semantic Approach. *Journal of Logic Programming*, 14:1–29, 1992.

[13] M. Bezem. Characterizing termination of logic programs with level mappings. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 69–80. The MIT Press, 1989.

[14] A. Bossi and N. Cocco. Verifying Correctness of Logic Programs. In J. Diaz and F. Orejas, editors, *TAPSOFT '89*, volume 352 of *Lecture Notes in Computer Science*, pages 96–110. Springer-Verlag, Berlin, 1989.

[15] A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and their use in Proving Universal Termination of a Logic Program. *Theoretical Computer Science*, 124:297–328, 1994.

[16] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The $\mathcal{S}$-semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19,20:149–197, 1994.

[17] K.A. Bowen and R.A. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 153–173. Academic Press, 1982.

[18] L. Cavedon. Acyclic logic programs and the completeness of SLDNF-resolution. *Theoretical Computer Science*, 86:81–92, 1991.

[19] K.L. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, Dept. of Computing, 1979.

[20] K.L. Clark and S.A. Tärnlund. A first order theory of data and programs. In *Information Processing '77*, pages 939–944. North Holland, 1977.

[21] L. Colussi and E. Marchiori. Proving correctness of logic programs using axiomatic semantics. In *Proceedings of the Eight International Conference on Logic Programming*, pages 629–644. The MIT Press, 1991.

[22] L. Colussi, E. Marchiori, and M. Marchiori. On Termination of Constraint Logic Programs. In M. Bruynooghe and J. Penjam, editors, *Proc. of PPCP'95*, number 976 in Lectures Notes in Computer Science, 1995.

[23] L. Crnogorac, A. D. Kelly, and H. Søndergaard. A Comparison of Three Occur-Check Analysers. In *Proc. of SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 159–173. Springer-Verlag, Berlin, 1996.

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

[25] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 8:69–116, 1987.

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

[27] W. Drabent and J. Maluszynski. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

[28] M. Ducasseé and J. Noyé. Logic Programming Environments: Dynamic Program Analysis and Debugging. *Journal of Logic Programming*, 19-20:351–384, 1994.

[29] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[30] M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative Modeling of the Operational Behaviour of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, December 1989.

[31] C.J. Hogger. *Introduction to Logic Programming.* Academic Press, London, 1984.

[32] M. Kalsbeek. Correctness of the Vanilla Meta-interpreter and Ambivalent Syntax. In K.R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 3–26. The MIT Press, 1995.

[33] K. Kunen. Some remarks on the completed database. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 978–992. The MIT Press, 1988.

[34] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, Berlin, Berlin, second edition, 1987.

[35] B. Malfon. Characterization of some semantics for logic programs with negation and application to program validation. In M. Bruynooghe, editor, *Proceedings of the International Symposium on Logic Programming*, pages 91–105. The MIT press, 1994.

[36] E. Marchiori and F. Teusink. On Termination of Logic Programs with Delay Declarations. In *Proc. of the 1995 International Symposium on Logic Programming*, pages 447–461. The MIT Press, 1995.

[37] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. on Programming Languages and Systems*, 4(2):258–282, 1982.

[38] F. Mesnard. Inferring Left-terminating Classes of Queries for Constraint Logic Programs. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 7–21. The MIT Press, 1996.

[39] L. Naish. Automatic Control for Logic Programs. *Journal of Logic Programming*, 2(3), 1985.

[40] L. Naish. Types and Intended Meaning. In F. Pfenning, editor, *Types in Logic Programming*, pages 189–216. MIT Press, Cambridge, Massachussets, 1992.

[41] D. Pedreschi and S. Ruggieri. Modular Verification of Logic Programs. In F. de Boer and M. Gabbrielli, editors, *Proc. of the W2 Post-Conference Workshop of the 1996 IJCSLP*, Technical Report TR-96-31. Dipartimento di Informatica, Università di Pisa, 1996.

[42] D. Pedreschi and S. Ruggieri. A Theory of Weakest Preconditions for Logic Programs. Unpublished, 1997.

[43] D. Pedreschi and S. Ruggieri. Verification of Metainterpreters. *Journal of Logic and Computation*, 7(2), April 1997.

[44] L. Plümer. *Termination Proofs for Logic Programs.* Lecture Notes in Artificial Intelligence 446, Springer-Verlag, Berlin, 1990.

[45] S. Ruggieri. Metodi formali per lo sviluppo di programmi logici. Master's thesis, Dipartimento di Informatica, Università di Pisa, 1994. Tesi di Laurea, in Italian.

[46] S. Ruggieri. Decidability of Logic Program Semantics and Applications to Testing. In *Proc. of PLILP'96*, volume 1140 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, Berlin, 1996.

[47] S. Ruggieri. On termination of constraint logic programs. In *Proceedings of the APPIA-GULP-PRODE'96 Joint Conference on Declarative Programming*, 1996.

[48] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.

[49] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analyzing the termination of definite logic programs with respect to call patterns. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 481–488. Institute for New Generation Computer Technology, 1992.

[50] J. Shoenfield. *Mathematical logic*. Addison Wesdley, Reading, 1967.

[51] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.

[52] J.D. Ulmann and A. van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345–373, 1988.

[53] T. Vasak and J. Potter. Characterization of terminating logic programs. In *Proceedings of the 1986 IEEE Symposium on Logic Programming*, 1986.

[54] B. Wang and R. K. Shyamasundar. A Methodology for Proving Termination of Logic Programs. *Journal of Logic Programming*, 21(1):1–30, 1994.

Appendix

A Termination

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

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

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 $bag(a_1, \ldots, a_n)$. 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

$$X \prec Y \text{ iff } X = Y - \{a\} \cup Z \text{ for a } a \in Y \text{ and } Z \text{ such that } b < a \text{ for every } b \in Z,$$

where X, Y, Z are finite multiset of elements from W. A well-known result (cfr., [25]) shows that if $(W, <)$ is a well-founded ordering, then the corresponding multiset ordering is well-founded as well.

In this appendix, we make use of an equivalent formulation of relation $\vdash_t$ for queries.

DEFINITION A.1 Consider a level mapping $|\ |$ and Pre, $Post$.

- To a query $Q = B_1, \ldots, B_n$ we associate n sets of natural numbers defined as follows, for $i \in [1, n]$:

  $$|Q|_i = \{\ |A_i|\ |\ A_1, \ldots, A_n \text{ is a ground instance of } Q \text{ and } Post \models A_1, \ldots, A_{i-1}\ \}.$$

- A query Q is called *bounded* (by $|\ |$ and $Post$) iff $|Q|_i$ is finite, for every $i \in [1,n]$.
- For $Q = B_1, \ldots, B_n$, we define the multiset $\|Q\|$ of natural numbers as follows:

$$\|Q\| \ = \ bag\,(max\ |Q|_1, \ldots,\ max\ |Q|_n).$$ □

Observe that if $\vdash \{Pre\}\ Q\ \{Post\}$ then, Q is bounded (by $|\ |$ and $Post$) if and only if $\vdash_t \{Pre\}\ Q\ \{Post\}$ holds (by using $|\ |$). 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 3.18.

LEMMA A.2 *Assume that* $\vdash_t \{Pre\}\ P\ \{Post\}$ *and* $\vdash_t \{Pre\}\ Q\ \{Post\}$ *hold by the same level mapping* $|\ |$. *Let* Q' *be a LD-resolvent of* Q *and* P. *Then*
(i) Q' *is bounded (by* $|\ |$ *and* $Post$*), and*
(ii) $\|Q'\| \prec_m \|Q\|$, *and*
(iii) $\vdash_t \{Pre\}\ Q'\ \{Post\}$.

Proof. *(i-ii)* The conclusions are consequences of the following facts, in which we refer to the level mapping and the interpretations of the enunciate.

FACT 1 *An instance* Q' *of a bounded query* Q *is bounded and* $\|Q'\| \preceq_m \|Q\|$.

Proof. It is sufficient to note that $|Q'|_i \subseteq |Q|_i$ for $i \in [1,n]$. □

FACT 2 *For every* $A \leftarrow \mathbf{B}$ *instance of a clause from* P, *and query* $\mathbf{C}$ *such that* $\vdash \{Pre\}\ A, \mathbf{C}\ \{Post\}$, *if* $A, \mathbf{C}$ *is bounded, then* $\mathbf{B}$,$\mathbf{C}$ *is bounded and* $\|\mathbf{B},\mathbf{C}\| \prec_m \|A, \mathbf{C}\|$.

Proof. Since $\vdash_t \{Pre\}\ P\ \{Post\}$ holds, we have by Definition 3.2 that $\vdash_t \{Pre\} A \leftarrow \mathbf{B} \{Post\}$ holds. Suppose now $\mathbf{B} = B_1, \ldots, B_n$ and $\mathbf{C} = C_1, \ldots, C_m$, for $n, m \geq 0$. Let us prove, first of all, the following facts.

NOTE 1 *For* $i \in [1,n]$, $|B_1, \ldots, B_n, C_1, \ldots, C_m|_i$ *is finite, and*

$$max|B_1, \ldots, B_n, C_1, \ldots, C_m|_i < max|A, C_1, \ldots, C_m|_1.$$

Proof. We have

$$\begin{array}{ll}
 & max|B_1, \ldots, B_n, C_1, \ldots, C_m|_i \\
= & \quad \{\text{ Definition A.1 }\} \\
 & max\{\ |B'_i| \mid B'_1, \ldots, B'_n \text{ is a ground instance of } \mathbf{B} \text{ and } Post \models B'_1, \ldots, B'_{i-1}\ \} \\
= & \quad \{\text{ for some } A',\ A' \leftarrow B'_1, \ldots, B'_n \text{ is a ground instance of } A \leftarrow \mathbf{B}\ \} \\
 & max\{\ |B'_i| \mid A' \leftarrow B'_1, \ldots, B'_n \text{ is a ground instance of } A \leftarrow \mathbf{B} \\
 & \text{and } Post \models B'_1, \ldots, B'_{i-1}\ \} \\
< & \quad \{\text{ Definition 3.2, } A \in Pre, \vdash_t \{Pre\}\ A \leftarrow \mathbf{B}\ \{Post\} \text{ and the fact that} \\
 & \quad \forall x \in S\ \exists y \in R : x < y \text{ implies } max\ S < max\ R\} \\
 & max\{\ |A'| \mid A' \text{ is a ground instance of } A\} \\
= & \quad \{\text{ Definition A.1 }\} \\
 & max|A, C_1, \ldots, C_m|_1.
\end{array}$$ □

NOTE 2 *For* $j \in [1,m]$, $|B_1, \ldots, B_n, C_1, \ldots, C_m|_{j+n}$ *is finite, and*

$$max|B_1, \ldots, B_n, C_1, \ldots, C_m|_{j+n} \leq max|A, C_1, \ldots, C_m|_{j+1}.$$

Proof. We have

$$
\begin{array}{ll}
 & max|B_1,\ldots,B_n,C_1,\ldots,C_m|_{j+n} \\
= & \quad \{ \text{ Definition A.1 } \} \\
 & max\{\ |C'_j| \mid B'_1,\ldots,B'_n,C'_1,\ldots,C'_m \text{ is a ground instance of } \mathbf{B},\mathbf{C} \\
 & \qquad \text{and } Post \models B'_1,\ldots,B'_n,C'_1,\ldots,C'_{j-1}\} \\
\leq & \quad \{ \text{ for some } A',\ A' \leftarrow B'_1,\ldots,B'_n \text{ is a ground instance of } A \leftarrow \mathbf{B}, \\
 & \ \text{and } \vdash \{Pre\}\ A \leftarrow \mathbf{B}\ \{Post\}\ , \\
 & \quad \text{and } S \subseteq R \text{ implies } max\ S \leq max\ R\} \\
 & max\{\ |C'_j| \mid A',C'_1,\ldots,C'_m \text{ is a ground instance of } A,\mathbf{C} \\
 & \qquad \text{and } PostA',C'_1,\ldots,C'_{j-1}\ \} \\
= & \quad \{ \text{ Definition A.1 } \} \\
 & max|A,C_1,\ldots,C_m|_{j+1}.
\end{array}
$$

□

As a consequence of Notes 1 and 2, $\mathbf{B},\mathbf{C}$ is bounded and

$$bag(max|\mathbf{B},\mathbf{C}|_1,\ldots,max|\mathbf{B},\mathbf{C}|_{n+m}) \prec_m bag(max|A,\mathbf{C}|_1,\ldots,max|A,\mathbf{C}|_{m+1})$$

and then the conclusion $\|\mathbf{B},\mathbf{C}\| \prec_m \|A,\mathbf{C}\|$. □

By hypothesis, $\vdash \{Pre\}\ Q\ \{Post\}$ holds and Q is bounded. Assume that θ is the mgu of Q and a clause C from P. Therefore, Fact 1 implies that $Q\theta$ is bounded and $\|Q\theta\| \prec_m \|Q\|$. Fact 2 considered with $A \leftarrow \mathbf{B} = C\theta$ and $A,\mathbf{C} = Q\theta$ implies that $\mathbf{B},\mathbf{C} = Q'$ is bounded and $\|Q'\| \prec_m \|Q\theta\| \preceq_m \|Q\|$.

(iii) By Theorem 3.18, $\vdash \{Pre\}\ Q'\ \{Post\}$ holds. Therefore, $\vdash_t \{Pre\}\ Q'\ \{Post\}$ holds since Q' is bounded as shown in *(i)*. □

In the rest of this section, our goal is to show the Termination Completeness Theorem 3.31. To this end, we denote by $nodes_P(Q)$ the number of nodes of a LD-tree for P and Q. The function $nodes_P$ is well-defined as two LD-trees have the same number of nodes. Next lemma reported from [4] will be useful.

LEMMA A.3 *Let P be a program and Q a query such that the LD-tree for P and Q is finite. Then*

(i) for all substitutions θ, $nodes_P(Q\theta) \leq nodes_P(Q)$,

(ii) for all prefixes H of Q, $nodes_P(H) \leq nodes_P(Q)$,

(iii) for all non-root nodes H in the LD-tree for P and Q $nodes_P(H) < nodes_P(Q)$.

Proof.

(i) By the Lifting Lemma (see Lloyd [34]), we can conclude that to every LD-derivation of P and $Q\theta$ with input clauses $C_1,C_2,\ldots$, there corresponds an LD-derivation for P and Q with variants of input clauses $C_1,C_2,\ldots$, of the same or larger length. This implies the claim.

(ii) Consider a prefix $H = B_1,\ldots,B_k$ of $Q = B_1,\ldots,B_n$ $(n \geq k)$. By an appropriate renaming of variables (formally justified by the Variant Lemma (see Apt [6])) we can assume that all input clauses used in the LD-tree for P and H have no variables in common with Q. We can now transform the LD-tree for P and H into an initial subtree of the LD-tree for P and Q by replacing in it a node $\mathbf{B}$ by $\mathbf{B},B_{k+1}\theta,\ldots,B_n\theta$, where θ is the composition of the mgu's used from the root H to the node $\mathbf{B}$. This implies the claim.

(iii) Immediate by the definition of $nodes_P$. □

We can now demonstrate the desired result.

THEOREM A.4 *Assume that* $\vdash \{Pre\}\ P\ \{Post\}$ *and* P *is Pre-terminating. Then there exists a level mapping* $|\ |$ *such that*
(i) $\vdash_t \{Pre\}\ P\ \{sp(P, Pre)\}$, *and*
(ii) for every query Q *such that* $\vdash \{Pre\}\ Q\ \{Post\}$, *we have that*

$$\vdash_t \{Pre\}\ Q\ \{sp(P, Pre)\}$$

holds by means of $|\ |$ *iff every LD-derivation for* P *and* Q *is finite.*

Proof. We recall that by Theorem 3.21, $sp(P, Pre) = M_P^L \cap Pre$. Consider now the level mapping $|\ |$ defined as

$$|A| = nodes_P(A).$$

for $A \in Pre$ and 0 elsewhere. The level mapping is well-defined since we assumed that P is Pre-terminating. Let us prove an half of (ii).

(ii⇐) We prove the conclusion by taking (cfr. Definition 3.2) $k = nodes_P(Q)$. We show that for every ground instance $A_1, \ldots, A_n$ of Q

$$\forall\ i \in [1, n]:\ M_P^L \cap Pre \models A_1, \ldots, A_{i-1} \Rightarrow k \geq |A_i|$$

Assume that $M_P \cap Pre \models A_1, \ldots, A_{i-1}$. We calculate

$$\begin{array}{lll}
k & = & nodes_P(Q) \\
& \geq & \{ \text{Lemma A.3 } \textit{(i)} \} \\
& & nodes_P(A_1, \ldots, A_n) \\
& \geq & \{ \text{Lemma A.3 } \textit{(ii)} \} \\
& & nodes_P(A_1, \ldots, A_i) \\
& \geq & \{ \text{Lemma A.3 } \textit{(iii)}, \text{ by noting that } M_P^L \cap Pre \models A_1, \ldots, A_{i-1} \\
& & \text{implies that there exists a LD-refutation for } P \text{ and } A_1, \ldots, A_{i-1} \} \\
& & nodes_P(A_i) \\
& = & \{ \text{Definition of } |\ | \} \\
& & |A_i|.
\end{array}$$

(i) Let us prove that $\vdash_t \{Pre\}\ P\ \{sp(P, Pre)\}$ holds by means of $|\ |$.
Given $A\theta \leftarrow B_1\theta, \ldots, B_n\theta \in ground_L(P)$, with $A \leftarrow B_1, \ldots, B_n$ clause from P, we have only to verify for $i \in [1, n]$

$$Pre \models A\theta\ \wedge\ M_P^L \cap Pre\ \models (A_1, \ldots, A_{i-1})\theta\ \Rightarrow |A\theta| > |B_i\theta|$$

First of all, observe that $A\theta\theta$ is a variant of $A\theta$, and then $A\theta$ and A unify. Consider $\mu \in mgu(A\theta, A)$. Then $\theta = \mu\delta$ for some δ. By the definition of LD-resolution, $B_1\mu, \ldots, B_n\mu$ is a LD-resolvent of $A\theta$. Therefore

$$\begin{array}{lll}
& & |A\theta| \\
& = & \{ \text{Definition of } |\ | \} \\
& & nodes_P(A\theta) \\
& > & \{ \text{Lemma A.3 (iii), } B_1\mu, \ldots, B_n\mu \text{ is a resolvent of } A\theta \} \\
& & nodes_P(B_1\mu, \ldots, B_n\mu) \\
& \geq & \{ \text{Half } \textit{(ii⇐)}, \text{ with } Q := B_1\mu, \ldots, B_n\mu \text{ and } A_i := B_i\theta \} \\
& & |B_i\theta|.
\end{array}$$

(ii⇒) By (i) and Theorem 3.26 all the LD-derivations of P and Q are finite. □

B General Programs

LEMMA B.1 *Assume that* $\vdash_t \{Pre\}\ P\ \{Post\}$ *holds. For every* $A \in Pre$*:*
(i) If the LDNF-tree for P *and* A *is finitely failed then* $Post \models \neg A$*.*
(ii) If there is a LDNF-refutation for P *and* A *then* $Post \models A$*.*

Proof. The proof is by induction on the rank and on the depth of the LDNF-tree and of the LDNF-refutation, respectively.

rank = 0)
depth = 0) *(i)* If $Post \models A$ then by 5.4*(ii)*

$$\exists A \leftarrow L_1, \ldots, L_n \in ground_L(P)\ .Post \models L_1, \ldots, L_n\ .$$

But this is impossible as the depth is 0 and then A does not unify with any head.
(ii) As A is an instance of a head of a unitary clause, by Definition 5.4 *(i2)* we have $Post \models A$.
depth > 0) *(i)* If $Post \models A$ then by Definition 5.4 *(ii)* there exists $C\theta = A \leftarrow L_1, \ldots, L_n \in ground_L(P)$ such that

$$Post \models L_1, \ldots, L_n \tag{10}$$

Moreover, every L_i is positive as the rank is 0. The resolvent of A and C has a finitely failed LD(NF)-tree, and then so it is for every instance of its, as $L_1, \ldots, L_n$. Therefore there exists a least $i \in [1, n]$ such that $L_1, \ldots, L_{i-1}$ has a refutation and L_i has a finitely failed LD(NF)-tree. By inductive hypothesis *((ii))*

$$Post \models L_1, \ldots, L_{i-1}$$

and then, as $\vdash_t \{Pre\}\ P\ \{Post\}$, $Pre \models L_i$. By inductive hypothesis on the depth $Post \not\models L_i$. This contradicts (10).
(ii) Consider the resolvent of A along the LD(NF)-refutation. Every literal in it is positive as the rank is 0. A ground instance $B_1, \ldots, B_n$ of it has a LD(NF)-refutation, and then by inductive hypothesis on the depth

$$Post \models B_1, \ldots, B_n$$

As $\vdash_t \{Pre\}\ P\ \{Post\}$, by Definition 5.4 *(i2)* and last statement we conclude $Post \models A$.

rank > 0)
depth = 0) Analogously as for rank 0.
depth > 0) *(i)* If $Post \models A$ then by Definition 5.4 *(ii)* there exists $C\theta = A \leftarrow L_1, \ldots, L_n \in ground_L(P)$ such that

$$Post \models L_1, \ldots, L_n \tag{11}$$

The resolvent of A and C has a finitely failed LDNF-tree, and then so it is for every instance of its, as $L_1, \ldots, L_n$. Therefore there exists a least $i \in [1, n]$ such that $L_1, \ldots, L_{i-1}$ has a refutation and L_i has a finitely failed LDNF-tree. Therefore by inductive hypothesis on *(i,ii)*

$$Post \models L_1, \ldots, L_{i-1}$$

and, as $\vdash_t \{Pre\}\ P\ \{Post\}$, we have $Pre \models A_i$ if $L_i = A_i$ or $L_i = \neg A_i$.

- if L_i is positive then the reasoning is the same as for rank 0,

- if $L_i = \neg A_i$ then by inductive hypothesis on *(ii)*

$$Post \models L_1 , \ldots , L_{i-1}$$

As L_i has a finitely failed LDNF-tree, i.e. A_i has a LDNF-refutation, by inductive hypothesis on *(i)* we have $Post \models A_i$, and then $Post \not\models L_i$. This contradicts (11).

(ii) Consider the resolvent of A along the LDNF-refutation. A ground instance $L_1 , \ldots , L_n$ of it has a LDNF-refutation, and then by inductive hypothesis on the depth every positive literal is in $Post$ and by inductive hypothesis on *(i)* every negative literal is not in $Post$, i.e.

$$Post \models L_1 , \ldots , L_n$$

As $\vdash_t \{Pre\}\ P\ \{Post\}$, by Definition 5.4 *(i2)* and last statement we conclude $Post \models A$.

□

We extend the notion of boundness to general queries by defining

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

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

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

(ii) $\|Q'\| \prec_m \|Q\|$, *and*

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

Proof. In the case that a positive literal is selected, we follow the same reasoning of Lemma A.2. 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 Q. It is immediate to verify *(i,ii)*. In addition, by Lemma B.1 $Post \models \neg A$ and then $\vdash_t \{Pre\}\ Q'\ \{Post\}$, i.e. *(iii)*. □