Università di Pisa
Dipartimento di Informatica

Technical Report : TR-96-23

Verification of Meta-interpreters

Dino Pedreschi, Salvatore Ruggieri

June 07, 1996

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

Verification of Meta-interpreters

Dino Pedreschi and **Salvatore Ruggieri**
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail: {pedre,ruggieri}@di.unipi.it

Abstract

A novel approach to the verification of meta-interpreters is introduced. We apply a general purpose verification method for logic programs, proposed in [28], to the case study of the `Vanilla` and other logic meta-interpreters. We extend the standard notion of *declarative correctness*, and design a criterion for proving correctness of meta-interpreters in a general sense, including amalgamated and reflective meta-interpreters.

The contribution of this 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 mentioned proof method.

We believe that the obtained results illustrate the broad applicability of the adopted verification principles.

Keywords: Meta-programming, logic programming, verification of logic programs, `Vanilla`.

Notes: This article has been accepted for publication on *Journal of Logic and Computation*, 1996.

1 Introduction

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 [12]), 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 behaviour. 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 this paper is to introduce a general criterion for reasoning about meta-interpreters, and demonstrating its applicability. The basic idea of applying a general purpose verification method for logic programs to the case study of the `Vanilla` meta-interpreter leads to several interesting results when applied to the general case. Our approach is based on the verification method introduced in [29, 28], which is a uniform proof method for proving partial and total correctness of logic and Prolog programs.

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 this 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 mentioned proof method. 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.

The paper is organized as follows. Section 2 introduces some fundamental notions on meta-interpreters, such as correctness, amalgamation, reflection, and extensions. Section 3 and 4 are a brief overview of semantics and a proof theory for general purpose verification of logic and Prolog programs. In Section 5, some basic results of the proof method are reported and applied to the `Vanilla`. The approach is generalized to a *general criterion* in Section 6, where in addition some applications are discussed, including the treatment of amalgamation, reflection, and extensions. In Section 7, termination properties are considered. Section 8 shows that the proof theory notions are language independent. Finally, in Section 9 it is shown how to reason on non-ground semantics, such as the least term model and the $\mathcal{S}$-semantics.

Notation and ambivalent syntax

A word on terminology is in order. Throughout the paper we use the standard notation of logic programming, as in [25, 3], 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 [24, 21]. For the moment, we only assume the restricted case of *atoms-as-terms* ambivalent syntax. In Section 6.2, we also consider the *terms-as-atoms* case.

We use queries instead of goals and 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.

LD-resolution is SLD-resolution with the leftmost selection rule. L_P 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. By $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ we denote a ground instance of a clause from P, and by $bag(a_1, \ldots, a_n)$ the multiset consisting of elements $a_1, \ldots, a_n$. 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: $T_P(I) = \{ A \in B_L \mid \exists A \leftarrow B_1, \ldots, B_n \in ground_L(P) \ : \ B_1, \ldots, B_n \in I \}$.

2 Meta-Interpreters

Correctness

What is a meta-program? Generally speaking, the answer is not obvious, as in computational logic programs and data are of the "same" nature. In this paper, we make the liberal assumption that meta-programs are just *programs* over the ambivalent language. It is however necessary to agree when a given meta-program can be regarded as a *meta-interpreter* in a meaningful sense, i.e., to define a notion of *correctness* of a meta-interpreter with respect to the object program under consideration. We stipulate that a meta-interpreter is a meta-program defining a specific relation called `demo`, which is expected to represent provability of object-level queries. This is the motivation for the following definition of correct meta-interpreters. For the sake of generality, the definition is parametric with respect to a set *Pre* of object-level atoms, which represents the intended (one-atom) object-level queries. We recall that L is the language in which programs and queries are written.

Definition 1 A program V is *declaratively correct* with respect to a program P and a set $Pre \subseteq B_L$ iff for every $A \in Pre$

$$A \in M_P^L \quad \text{iff} \quad \mathtt{demo}(A) \in M_V^L$$

When the *if* part only is considered, we have a notion of *soundness*. In this case, we say that V is *declaratively sound* w.r.t. P and *Pre*. □

The intuition is that, when considering *intended* queries only, the `demo`-predicate of a correct meta-interpreter behaves in the same way as the object-level program. In the literature, the case $Pre = B_P$ is usually considered, i.e., all object-level queries are considered in the above definition of correctness.

Vanilla

A *jewel* of Logic Programming is the elegant meta-program which, by means of the ambivalent syntax, specifies the meta-circular interpreter for LP (i.e., the interpreter of LP in LP). This program, referred to as the `Vanilla` meta-interpreter, and first introduced in [12], is denoted by $Van(P)$ when instantiated with an object program P, and consists of the following clauses:

$(d1)$ `demo( true ).`

$(d2)$
```
demo( A & B ) ←
   demo( A ),
   demo( B ).
```

$(d3)$
```
demo( A ) ←
   clause( A , B ),
   demo( B ).
```

(c) `clause( A , B`$_1$ `& ... & B`$_n$ `).` for every `A` $\leftarrow$ `B`$_1$`, ..., B`$_n \in P$

where `B`$_1$ `& ... & B`$_n$ is an abbreviation for

- `B`$_1$ `& ( B`$_2$ `& ... ( B`$_{n-1}$ `& B`$_n$ `) ... )` , if $n > 1$,
- `B`$_1$, if $n = 1$
- `true` , if $n = 0$.

For instance, for $P =$ `APPEND` the definition of `clause` is

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

Specializing the above notion of declarative correctness, we have that $Van(P)$ is *declaratively correct* w.r.t. P and *Pre* iff for every $A \in Pre$ we have $A \in M_P^L$ iff `demo(`A`)` $\in M_{Van(P)}^L$.

It is important to require that the symbols `&` and `true` are not predicate symbols of L, i.e., object-level predicates. Indeed, without this assumption the `Vanilla` is not correct. Consider, in fact, the program consisting of the three unit clauses `q(a).`, `p(b).` and `&(p(c),p(c)).` Then `demo(&(p(b),q(a)))` is in $M_{Van(P)}^L$ whereas the atom `&(p(b),q(a))` is not in M_P^L. A similar argument applies to the program `true` $\leftarrow$ `true`.

The relevance of `Vanilla` lies in the fact that it provides a terse account of the provability relation of logic programming, and a natural basis for extending the computation strategy. It should be noted that a meta-interpreter such as `Vanilla`, which basically relies on the ambivalent syntax, cannot be defined in strongly typed languages, such as Gödel [20], as in every type discipline functors and relations cannot be confused. Due to its unique nature, the `Vanilla` meta-interpreter has raised non-trivial questions concerning its correctness, and a vast literature on the subject is available [8]. We propose in this paper a systematic method for assessing the correctness of `Vanilla` based on its declarative semantics, and improve on existing results in various directions. The advantages of the method introduced in the present paper can be summed up as follows. First, our method yields simpler correctness proofs. Second, new results are established as a by-product. Third, a correctness criterion for arbitrary meta-interpreters is introduced, as a result of the analysis of `Vanilla`.

Amalgamation and Reflection Down/Up

The ambivalent syntax allows us to combine the object program with the meta-program - a technique known as *amalgamation* [12]. We will show that, under certain assumptions, `Vanilla` can be amalgamated with the object program without compromising the declarative correctness.

According to the treatment of amalgamation reported in [22, 26], we will consider several types of amalgamation. A simple form of amalgamation is joining object and meta- program preventing any interference. A more liberal approach allows the object-program to make a call to the meta-level. This possibility is usually referred to as *reflection up*. As an example, we consider `Vanilla` augmented with the clause:

```
demo( demo(A) ) ← demo( A ).
```

which allows the object program to make a call at the meta-level.

On the other hand, in the form of amalgamation known as *reflection down*, it is the meta-program which makes a call to the object-level. The simplest reflective meta-interpreter is the standard Prolog implementation of the meta-predicate `call` [31] by means of meta-variables:

```
call( X ) ← X.
```

In order to admit such a program, we need a more complex form of ambivalent syntax, called *terms-as-atoms* (see [24].)

Parameterization and Extensions

Meta-Interpreters such as `Vanilla` may be enhanced with various functionalities in a simple way. For example, the `demo` relation can be augmented with new arguments which represent proof trees and other information about proofs (see [31] for a collection of enhanced meta-interpreters). Also, extra arguments may be used to deal with theories or modules, thus supporting various forms of modular programming and program composition [15]. Later we discuss how the proposed approach may be extended to deal with enhanced meta-interpreters.

3 Verification

Logic programming (and Prolog) is advertised as a *declarative* language, in the sense that *specifications*, when written in an appropriate syntax, can be directly used as *programs*. This ideal situation, however, is often unrealistic in practice, both for the fact that executable specifications may be extremely inefficient, and for problems related to the implementations of logic programming—programs may fail to terminate or may result in errors when executed with particular strategies.

It is therefore needed to assess the correctness of a logic or Prolog programs with respect to its specification, or *intended interpretation*: a task that has been underestimated in the literature until recently. Of course, when verifying a logic program P, it would be helpful to use its declarative semantics, namely its *least Herbrand model* M_P^L. A natural approach is to consider 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 model 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 [31]:

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 [28, 29].

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

$$\begin{array}{rcl} Pre & = & \{\ \texttt{append(}xs\texttt{, } ys\texttt{, } zs\texttt{)} \mid xs, ys \text{ are lists }\} \\ Post & = & \{\ \texttt{append(}xs\texttt{, } ys\texttt{, } zs\texttt{)} \mid zs \text{ is the concatenation of the lists } xs \text{ and } ys\ \}. \end{array}$$

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 weakly totally correct w.r.t. $Pre, Post'$, where Pre is as above, and

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

where $|.|$ denotes the length of a list. Therefore, a postcondition in the sense of the weak correctness describes a property of every computed 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 requiring *existential termination* of the atoms in the precondition w.r.t. a fair selection rule and a complete search strategy. 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.

4 Proof Theory

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.

Throughout the paper, we consistently adopt a format of presentation consisting of the following steps

- the proof techniques are introduced first,
- a simple example is directly provided,
- the key results about the proof technique are reported, possibly pointing to appropriate references,
- finally, the application to the various versions of the `Vanilla` meta-interpreter is described.

4.1 The proof method

The next crucial definition explains the proof obligations needed to prove that the relations $\vdash$ or $\vdash_t$ hold on a given 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 [9] the notion of a *level mapping* (on a language L) which is a function $|\ |: B_L \to N$ from ground atoms to natural numbers. Level mappings play the role of termination functions in the proofs of total correctness.

Definition 3 Consider a program P and a query Q. Given 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 $i \in [1, n]$ $\quad Post \models A_1, \ldots, A_{i-1} \Rightarrow Pre \models A_i$

- $\vdash_t \{Pre\}\ Q\ \{Post\}$ 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 $i \in [1,n]$ $\quad Post \models A_1, \ldots, A_{i-1} \Rightarrow Pre \models A_i \quad \wedge \quad k > |A_i|$ $\square$

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 are 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. We shall see in the sequel that the notion $\vdash_t \{Pre\}\ Q\ \{Post\}$ is a tool for proving that the query Q universally terminates.

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.

The relation $\vdash$ for (weak) partial correctness is equivalent to the method proposed in [10]. However, our method can be applied in a simpler way in that it involves reasoning at the level of ground instances, whereas the method in [10] considers arbitrary instances of the program clauses. The relation $\vdash_t$ for (weak) total correctness is a generalization of the method for proving termination in [7, 2]. The proposed method combines the two approaches [10] and [7] into a unified one, which thus provides a uniform framework for addressing the various verification issues. The mentioned method in [10] has been also extended in [11] to deal with termination, by adopting level mappings on non-ground atoms. This leads to complications, since restrictions are to be imposed on the level mappings, which are prevented in our approach.

There are several symmetries in the Definitions 3. In particular, we want to point out that the Definitions 3 *(3,4)* for a query Q derive from Definitions 3 *(1,2)* for the program $goal \leftarrow Q$ where $goal$ is a new predicate. This follows from the following useful relation, which is immediate from Definition 3. For $\vdash \{Pre\}\ P\ \{Post\}$ and $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$,

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

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

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

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 *(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, $Post$.

Consider, as an example, the following proof outline for the recursive clause of `APPEND`.

$\{$ `[`X`|`Xs`]` and Ys lists $\}$
`append( [X|Xs], Ys, [X|Zs] )` $\leftarrow$ $\qquad\qquad \{\ |Xs| + 1\ \}$
$\qquad \{\ Xs$ and Ys lists $\}$

```
append( Xs, Ys, Zs ).                    { |Xs| }
  { Zs is the concatenation of Xs and Ys }
{ [X|Zs] is the concatenation of [X|Xs] and Ys }
```

where *Pre* and *Post* are chosen as in Section 3, and the level mapping for an **append** atom is chosen as the length $|Xs|$ of its first argument Xs. It is immediate to verify that the proof obligations are satisfied.

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.

In the sequel, we shall also make use of a weakened 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.

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

Similarly, the proof obligations (ii,iii,v) imply condition *2* of Definition 3.

Vanilla

As a first exercise, let us show that the relation $\vdash$ lifts from the object program up to **Vanilla**, in the sense that if $\vdash \{Pre\}\ P\ \{Post\}$ holds, then $\vdash \{Pre_{Van(P)}\}\ Van(P)\ \{Post_{Van(P)}\}$ holds for certain $Pre_{Van(P)}$, $Post_{Van(P)}$ defined starting from *Pre* and *Post* in a natural way. (Weak) total correctness of **Vanilla** connected to the $\vdash_t$ relation, is discussed in Section 7.

Let $B_1, \ldots, B_n$ be ground atoms, with $n \geq 0$. We recall that B_1 **&** $\ldots$ **&** B_n is an abbreviation for

- $(B_1$ `&` $($ B_2 `&` $\ldots(B_{n-1}$ `&` $B_n)\ldots))$, if $n > 1$,
- B_1, if $n = 1$
- and `true` , if $n = 0$.

Conversely, by writing $(B_1$ `&` $\ldots$ `&` $B_n)^-$ we denote the query $B_1, \ldots, B_n$.
We define:

$$\begin{array}{rll} \texttt{demo(}B_1 \texttt{ \& } \ldots \texttt{ \& } B_n\texttt{)} \in Pre_{Van(P)} & \text{iff} & \vdash \{Pre\}\ B_1, \ldots, B_n\ \{Post\} \\ \texttt{clause(}A\texttt{, }B\texttt{)} \in Pre_{Van(P)} & \text{iff} & true \end{array}$$

$$\begin{array}{rll} \texttt{demo(}B_1 \texttt{ \& } \ldots \texttt{ \& } B_n\texttt{)} \in Post_{Van(P)} & \text{iff} & Post \models B_1, \ldots, B_n \\ \texttt{clause(}A\texttt{, }B_1 \texttt{ \& } \ldots \texttt{ \& } B_n\texttt{)} \in Post_{Van(P)} & \text{iff} & A \leftarrow B_1, \ldots, B_n \in ground_L(P) \end{array}$$

No other atom is in $Pre_{Van(P)}$ or in $Post_{Van(P)}$. Formally, $Pre_{Van(P)}$ and $Post_{Van(P)}$ are defined in terms of Pre and $Post$, and their definition reflects at the meta-level the notions of intended queries and intended correct instances of queries at the object-level. The next proof outlines establish that $\vdash \{Pre_{Van(P)}\}\ Van(P)\ \{Post_{Van(P)}\}$.

(a)
$\{\ true\ \}$
`demo(true).`
$\{\ true\ \}$

(b)
$\{\ \vdash \{Pre\}\ A, B^-\ \{Post\}\ \}$
`demo(A & B ) ←`
  $\{\ Pre \models A\ \}$
  `demo(A),`
  $\{\ Post \models A\ \}$
  $\{\ \vdash \{Pre\}\ B^-\ \{Post\}\ \}$
  `demo(B).`
  $\{\ Post \models B^-\ \}$
$\{\ Post \models A, B^-\ \}$

(c)
$\{\ \vdash \{Pre\}\ A\ \{Post\}\ \}$
`demo(A) ←`
  $\{\ true\ \}$
  `clause(A, B),`
  $\{\ A \leftarrow B^- \in ground_L(P)\ )\ \}$
  $\{\ \vdash \{Pre\}\ B^-\ \{Post\}\ \}$
  `demo(B).`
  $\{\ Post \models B^-\ \}$
$\{\ Post \models A\ \}$

(d)
$\{\ true\ \}$
`clause(A, B).`
$\{\ A \leftarrow B^- \in ground_L(P)\ \ \}$

Proof outlines *(a,b)* are self-explanatory, by simply observing that for a ground query $B_1, \ldots, B_n$:

(i) $\vdash \{Pre\}\ B_1, \ldots, B_n\ \{Post\} \quad \Rightarrow \quad Pre \models B_1$

(ii) $\vdash \{Pre\}\ B_1, \ldots, B_n\ \{Post\} \wedge Post \models B_1 \quad \Rightarrow \quad \vdash \{Pre\}\ B_2, \ldots, B_n\ \{Post\}$

The proofs of *(i,ii)* are immediate by the Definition 3. Proof outline *(d)* is actually a proof outline *schema* (one for each clause from P), and it is of direct verification.

Consider now the proof outline *(c)*. First, we observe that, as a direct consequence of Definition 3*(3)*, we have for an atom A:

$$\vdash \{Pre\}\ A\ \{Post\} \quad \Leftrightarrow \quad Pre \models A \tag{2}$$

Let us prove the proof obligations (iv) case $i = n$, and (v). To establish the proof obligation (iv) (case $i = 2$), assume $\vdash \{Pre\}\ A\ \{Post\} \wedge A \leftarrow B^- \in ground_L(P)$. By (2), we have $Pre \models A$ and, as observed in (1), we conclude $\vdash \{Pre\}\ B^-\ \{Post\}$.

To establish the proof obligation (v), assume $\vdash \{Pre\}\ A\ \{Post\} \wedge A \leftarrow B^- \in ground_L(P) \wedge Post \models B^-$. By (2) and the last two conjuncts we conclude directly by Definition 3 that $Post \models A$.

5 Partial Correctness

5.1 Weak Partial Correctness

In this section, we recall some properties of the relation $\vdash$ (see, e.g., [28] or [6]) and apply them to our case study. First, a justification is provided of the intuitive fact that the postcondition is a description of the correct instances of the queries which satisfy the relation $\vdash \{Pre\}\ Q\ \{Post\}$.

Lemma 5 Let P be a program and Q a query such that $\vdash \{Pre\}\ P\ \{Post\}$ and $\vdash \{Pre\}\ Q\ \{Post\}$. Then:

(i) for every SLD-resolvent Q' of Q and P the relation $\vdash \{Pre\}\ Q'\ \{Post\}$ holds, and

(ii) for every computed (or correct) instance Q' of Q and P we have $Post \models Q'$. □

As a direct consequence, the relation $\vdash$ is a proof method for weak partial correctness, i.e. $\vdash \{Pre\}\ P\ \{Post\}$ implies $M_P^L \cap Pre \subseteq Post$. We will provide later a strengthening of the method to prove the other inclusion, namely partial correctness.

Theorem 6 (Weak Partial Correctness) If $\vdash \{Pre\}\ P\ \{Post\}$ holds, then P is weak partially correct w.r.t. the specification $(Pre, Post)$.

Proof Consider $A \in M_P^L \cap Pre$. Then $\vdash \{Pre\}\ A\ \{Post\}$ holds, and, by Completeness of SLD-resolution, the query A has an SLD-refutation. So $A \in Post$ by Lemma 5 (ii). □

Vanilla: Weak declarative soundness and Call patterns

As we proved that $\vdash \{Pre_{Van(P)}\}\ Van(P)\ \{Post_{Van(P)}\}$ holds, under the hypothesis that $\vdash \{Pre\}\ P\ \{Post\}$ holds, Theorem 6 allows us to state that $Van(P)$ is weak partially correct w.r.t. the specification $(Pre_{Van(P)}, Post_{Van(P)})$ when $\vdash \{Pre\}\ P\ \{Post\}$.

In addition, considering a (not necessarily ground) query $B_1, \ldots, B_n$ such that

$$\vdash \{Pre\}\ B_1, \ldots, B_n\ \{Post\}$$

we obtain directly from the definition of $Pre_{Van(P)}$ that

$$\vdash \{Pre_{Van(P)}\}\ \texttt{demo(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{)}\ \{Post_{Van(P)}\} \tag{3}$$

Although Lemma 5 is a rather simple result, stating the persistency of $\vdash$ in SLD-derivations, it allows us to prove two natural properties of `Vanilla`, which are not reported in the literature.

First, we introduce the notion of weak declarative soundness of a meta-program, as a weakening of Definition 1.

Definition 7 A program V is *weak declaratively sound* with respect to a program P and a specification $(Pre, Post)$ iff for every $A \in Pre$:

$$A \in Post \quad \text{if} \quad \texttt{demo(}A\texttt{)} \in M_V^L$$

□

In the special case $Post = M_P^L$ we have again declarative soundness. Here, we only require that if $\texttt{demo(}A\texttt{)} \in M_V^L$ then A satisfies some property denoted by $Post$. This may be sufficient in many interesting cases, when some weak properties of the meta-interpreter are needed. Weak declarative soundness of $Van(P)$ is a simple consequence of the following theorem.

Theorem 8 Suppose that $\vdash \{Pre\}\ P\ \{Post\}$ and $\vdash \{Pre\}\ Q^-\ \{Post\}$. Then for every computed (correct) instance $\texttt{demo(}Q_1\texttt{)}$ of $Van(P)$ and $\texttt{demo(}Q\texttt{)}$, we have $Post \models Q_1^-$.

Proof As $\vdash \{Pre\}\ P\ \{Post\}$, we have $\vdash \{Pre_{Van(P)}\}\ Van(P)\ \{Post_{Van(P)}\}$; and as $\vdash \{Pre\}\ Q^-\ \{Post\}$, (3) implies that $\vdash \{Pre_{Van(P)}\}\ \texttt{demo(}Q\texttt{)}\ \{Post_{Van(P)}\}$ holds. Therefore, by Lemma 5 (ii), $Post_{Van(P)} \models \texttt{demo(}Q_1\texttt{)}$. The conclusion follows from the definition of $Post_{Van(P)}$. □

As a consequence of this simple result we have that the properties of correct instances lifts up from the object to the meta-level in the case of the `Vanilla` meta-interpreter. In other words, `Vanilla` preserves the declarative properties of object-level correct instances of intended queries.

The second relevant consequence of Lemma 5 is the persistency of call patterns from the object program to `Vanilla`, under the leftmost selection rule of Prolog.

Theorem 9 Suppose that $\vdash \{Pre\}\ P\ \{Post\}$ and $\vdash \{Pre\}\ Q^-\ \{Post\}$. Then for every selected atom $\texttt{demo(}Q_1\texttt{)}$ in a LD-derivation for $Van(P)$ and $\texttt{demo(}Q\texttt{)}$, we have $\vdash \{Pre\}\ Q_1^-\ \{Post\}$.

Proof As $\vdash \{Pre\}\ P\ \{Post\}$, we have $\vdash \{Pre_{Van(P)}\}\ Van(P)\ \{Post_{Van(P)}\}$; and as $\vdash \{Pre\}\ Q^-\ \{Post\}$, (3) implies that $\vdash \{Pre_{Van(P)}\}$ `demo(`Q`)` $\{Post_{Van(P)}\}$ holds. Therefore, by Lemma 5 (i), $Pre_{Van(P)} \models$ `demo(`Q_1`)`. The conclusion follows from the definition of $Pre_{Van(P)}$. □

When Q_1 is an atom, the above result states that $Pre \models Q_1$. In this sense, call patterns of the object program lift up to `Vanilla`.

As an application, let us consider an extension of `Vanilla` which includes the treatment of arithmetic built-in's, such as $>$, obtained adding the clause

```
demo( X > Y ) ←X > Y.
```

According to [4], we stipulate that a computation *ends in an error* when an atom $s > t$ is selected and s, t are not ground arithmetic expressions, (in short gae's). By defining the precondition Pre as follows

$$s > t \in Pre \quad \text{iff} \quad s, t \text{ are gae's}$$

we have, in the hypothesis of Lemma 5 (i), that no LD-derivation ends in an error. Such a property is inherited by the precondition $Pre_{Van(P)}$ of `Vanilla`. Indeed, we have

$$\texttt{demo(}s\ \texttt{>}\ t\texttt{)} \in Pre_{Van(P)} \quad \text{iff} \quad s, t \text{ are gae's}$$

and then, in the hypothesis of Lemma 5 (i), no LD-derivation ends in an error. In this sense, we conclude that `Vanilla` extended with arithmetic built-in's, preserves the property of absence of errors, in the sense that the meta-program is error-free if the object program is.

5.2 Partial Correctness

The intuition underlying an *Hoare style* proof method based on pre- and postconditions becomes more concrete in the present context when dealing with modular proofs. The following modularity rule reflects an obvious property of triples:

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

The importance of this fact is twofold. On the one hand, it allows us to split a correctness proof into simpler ones.
On the other hand, it allows us to introduce a notion of *strongest postcondition*.

Definition 10 Let P be a program such that $\vdash \{Pre\}\ P\ \{Post\}$. By $sp(P, Pre)$ we denote the intersection of all $Post'$ such that $\vdash \{Pre\}\ P\ \{Post'\}$. □

This definition, however, is not very useful to construct $sp(P, Pre)$. We provide an alternative characterization, by relating $sp(P, Pre)$ to $M_P^L \cap Pre$. As $sp(P, Pre)$ is an intersection of postconditions, each of which includes $M_P^L \cap Pre$ by Theorem 6, we have the inclusion $sp(P, Pre) \supseteq M_P^L \cap Pre$. The reverse inclusion can be proved by observing that the interpretation $M_P^L \cap Pre$ is a postcondition if $\vdash \{Pre\}\ P\ \{Post\}$ holds. This fact is an immediate consequence of the inclusion $M_P^L \cap Pre \subseteq Post$ implied by Theorem 6. Thus, $M_P^L \cap Pre$ is indeed the strongest postcondition as recorded in the following result.

Theorem 11 For a program P such that $\vdash \{Pre\}\ P\ \{Post\}$ holds, we have that $\vdash \{Pre\}\ P\ \{M_P^L \cap Pre\}$ holds. Hence, $sp(P, Pre) = M_P^L \cap Pre$. □

As a result we have the following Partial Correctness Theorem.

Theorem 12 (Partial Correctness) A program P such that $\vdash \{Pre\}\ P\ \{Post\}$ is partially correct w.r.t. the specification $(Pre, sp(P, Pre))$. □

The problem is how to characterize the strongest postcondition without first constructing the least Herbrand model M_P^L. To this end, we introduce the notion of a *well-supported interpretation*, which are well-studied in the context of semantics for general logic programs (see [1].)

Definition 13 Suppose that $\vdash \{Pre\}\ P\ \{Post\}$.
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 entails that indeed M_P^L coincides with a well-supported interpretation on the atoms in *Pre*. Observe that, in Definition 13, 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.
The next result states that well-supported postconditions provide a proof method for partial correctness, which is complete for programs satisfying the relation $\vdash$. A proof of this result can be found in [28].

Theorem 14 Let P be a program such that $\vdash \{Pre\}\ P\ \{Post\}$. Then

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

□

Observe that if $\vdash \{Pre\}\ P\ \{Post\}$ holds then, by Theorem 11, $\vdash \{Pre\}\ P\ \{sp(P, Pre)\}$ holds. This and the fact that $sp(P, Pre) \cap Pre = sp(P, Pre)$ imply that $sp(P, Pre)$ is well-supported by the above Theorem.

Proof Outlines

Proving weak partial correctness is handy, as the task can be carried out using the proof outlines. In contrast, Definition 13 may seem intricate and difficult to handle with simple tools. Fortunately, it has a straightforward interpretation in terms of proof outlines.

Definition 15 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 labelled clause of the form

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

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 a set of 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.

Vanilla

We are now in position to show that $Post_{Van(P)}$ is a well-supported interpretation (w.r.t. `Vanilla` and $Pre_{Van(P)}$) when the postcondition $Post$ of the object program is well-supported.

Suppose $Post$ is well-supported. Then 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

$$\forall i \in [1, n] : Post \models B_i \wedge |A| > |B_i| \tag{4}$$

In order to construct the relevant proof outlines, consider the well-founded poset $(bag(W), \prec)$ over finite multiset of W induced by $(W, <)$, and the function

$$||\texttt{demo(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{)}|| = bag(|B_1|, \ldots, |B_n|)$$

if $\texttt{demo(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{)} \in Pre_{Van(P)}$ for $n \geq 1$, and $bag()$ otherwise.
Notice that for $\texttt{demo(}A\ \texttt{\&}\ B\texttt{)} \in Pre_{Van(P)}$

$$||\texttt{demo(}A\texttt{)}||, ||\texttt{demo(}B\texttt{)}|| \prec ||\texttt{demo(}A\ \texttt{\&}\ B\texttt{)}||. \tag{5}$$

the next proof outlines show that $Post_{Van(P)}$ is a well-supported interpretation.

	$\{\ true\ \}$	
(a)	`demo(true).`	$\{\ bag()\ \}$
	$\{\ \texttt{demo(}A\ \texttt{\&}\ B\texttt{)} \in Pre_{Van(P)}\ \wedge\ Post \models A, B^-\ \}$	
(b)	`demo(A & B)` $\leftarrow$	$\{\ ||\texttt{demo(}A\ \texttt{\&}\ B\texttt{))}||\ \}$
	`demo(A),`	$\{\ ||\texttt{demo(}A\texttt{)}||\ \}$
	$\{\ Post \models A\ \}$	
	`demo(B).`	$\{\ ||\texttt{demo(}B\texttt{)}||\ \}$
	$\{\ Post \models B^-\ \}$	
	$\{\ A \in Post \cap Pre\ \wedge\ Post \models B^-\ \wedge\ A \leftarrow B^- \in ground_L(P)\ \wedge$ $\wedge\ ||\texttt{demo(}B\texttt{)}||\ \prec ||\texttt{demo(}A\texttt{)}||\ \}$	
(c)	`demo(A)` $\leftarrow$	$\{\ ||\texttt{demo(}A\texttt{)}||\ \}$
	`clause(A , B),`	$\{\ bag()\ \}$
	$\{\ A \leftarrow B^- \in ground_L(P)\ \}$	
	`demo(B).`	$\{\ ||\texttt{demo(}B\texttt{)}||\ \}$
	$\{\ Post \models B^-\ \}$	
	$\{\ A \leftarrow B^- \in ground_L(P)\ \}$	
(d)	`clause(A, B).`	$\{\ bag()\ \}$

The proof outlines are of immediate verification by using the definitions of $Pre_{Van(P)}$ and $Post_{Van(P)}$.

To conclude that $Post_{Van(P)}$ is a well-supported interpretation, we have to show that every $A \in Post_{Van(P)} \cap Pre_{Van(P)}$ is a ground instance of some head atom in a proof outline and the assertion associated with the head holds.

For the atoms whose predicate symbol is `clause` the conclusion is trivial. Assume now that $\texttt{demo(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{)} \in Pre_{Van(P)} \cap Post_{Van(P)}$.

If $n \neq 1$ we use the proof outline *(a)* or *(b)*, and observe that the assertion of the head is the hypothesis $\texttt{demo(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{)} \in Pre_{Van(P)} \cap Post_{Van(P)}$..

If $n = 1$ then we consider proof outline *(c)*. We have to prove that for $A \in Post \cap Pre$ there exists B^- such that

$$Post \models B^- \wedge A \leftarrow B^- \in ground_L(P)\ \wedge ||\texttt{demo(}B\texttt{)}||\ \prec ||\texttt{demo(}A\texttt{)}||.$$

Such a B^- clearly exists as a consequence of (4).

5.3 Correctness of Vanilla

We are now in position to summarize the various results obtained so far into a proof of correctness of `Vanilla`.

Theorem 16 Suppose that $\vdash \{Pre\}\ P\ \{Post\}$. Then $Van(P)$ is declaratively correct with respect to P and Pre.

Proof By Theorem 11 we have $\vdash \{Pre\}\ P\ \{sp(P, Pre)\}$.

By defining $Pre_{Van(P)}, Post_{Van(P)}$ starting from $Pre, sp(P, Pre)$ we showed that $\vdash \{Pre_{Van(P)}\}\ Van(P)\ \{Post_{Van(P)}\}$ holds. Moreover, as by Theorem 14 $sp(P, Pre)$ is well-supported, we showed that $Post_{Van(P)}$ is well-supported as well, and then by Theorems 14 and 11

$$sp(Van, Pre_{Van(P)}) = Post_{Van(P)} \cap Pre_{Van(P)} = M^L_{Van(P)} \cap Pre_{Van(P)} \tag{6}$$

Considering $A \in Pre$, from which by definition $\texttt{demo}(A) \in Pre_{Van(P)}$, we have the desired conclusion, as follows

$$\begin{array}{ll} & \texttt{demo}(A) \in M^L_{Van(P)} \\ \Leftrightarrow & \quad \{\ (6) \wedge \texttt{demo}(A) \in Pre_{Van(P)}\ \} \\ & \texttt{demo}(A) \in Post_{Van(P)} \\ \Leftrightarrow & \quad \{\ \text{Definition of } Post_{Van(P)}\ \} \\ & A \in sp(P, Pre) \\ \Leftrightarrow & \quad \{\ \text{Theorem 11 and } A \in Pre\ \} \\ & A \in M^L_P. \end{array}$$

□

The same reasoning applies to the case of (not necessarily atomic) ground queries.

Corollary 17 For a program P and a ground query Q^- such that $\vdash \{Pre\}\ P\ \{Post\}$ and $\vdash \{Pre\}\ Q^-\ \{Post\}$ hold, we have

$$P \models Q^- \quad \text{iff} \quad Van(P) \models \texttt{demo}(Q).$$

□

As for any logic program P we have $\vdash \{B_L\}\ P\ \{B_L\}$, the last result implies that for every program P, a ground query Q is a logical consequence of P iff $\texttt{demo}(Q)$ is a logical consequence of `Vanilla` instantiated by P. Therefore, as a particular case of Corollary 17, we reconstruct the correctness theorem generally studied in the literature, simply noting that $B_P = B_{L_P} \subseteq B_L$ for the assumption that L extends L_P. We recall here the assumption that `&` and `true` are not predicate symbols of L, and a fortiori of L_P.

Corollary 18 For a program P and $A \in B_P$ we have

$$A \in M^L_P \quad \text{iff} \quad \texttt{demo}(A) \in M^L_{Van(P)}.$$

□

6 Proving declarative correctness

Can we generalize the method used to verify `Vanilla` to arbitrary meta-interpreters ? The answer to this question is affirmative, and this section is devoted to illustrate a general method to tackle the proof of correctness of a meta-interpreter. We stress that this method was literally extracted from the verification of the `Vanilla` in a systematic way. As an illustration, we apply it to several modifications and extensions of `Vanilla`, supporting amalgamation, modular programming and non-standard mechanisms.

6.1 A general method

The reasoning followed in the proof of correctness of the `Vanilla` can be easily generalized. The next definition provides us with a general criterion of correctness. It will be the starting point for a number of generalizations and extensions to non-standard meta-interpreters.

Definition 19 Suppose $\vdash \{Pre\}\ P\ \{Post\}$ with $Post$ well-supported w.r.t. P and Pre. A program V satisfies the General Criterion [GC] with respect to P and Pre iff, for some Pre', $Post'$, the following properties hold:

1. $\vdash \{Pre'\}\ V\ \{Post'\}$,

2. $Post'$ well-supported w.r.t. V and Pre',

3. $\forall A \in Pre.\quad \mathtt{demo}(A) \in Pre'$,

4. $\forall A \in Pre.\quad A \in Post \Leftrightarrow \mathtt{demo}(A) \in Post'$. □

Conditions *(3,4)* constrain Pre', $Post'$ to the case where the argument of `demo` belongs to Pre.

It is worth noting that neither M_V^L nor M_P^L appear in the formulation of the General Criterion. This fact makes it of practical interest. If the two set are known, we would not need any criterion. We could just compare them to check correctness.

Let us explain the intuition behind [GC]. Suppose we want to prove the declarative correctness of the program V with respect to an object program P such that $\vdash \{Pre\}\ P\ \{Post\}$ holds with $Post$ well-supported. If we can now prove *(3-4)* we have for $A \in Pre$

$$A \in Post \cap Pre \Leftrightarrow \mathtt{demo}(A) \in Post' \cap Pre'$$

and, as $Post$ (by hypothesis) and $Post'$ (by *(2)*) are well-supported, by Theorems 14 and 11 (using *(1,2)*)

$$A \in M_P^L \Leftrightarrow \mathtt{demo}(A) \in M_V^L$$

To prove [GC], it is sufficient to assume that $Post$ is well-supported, without necessarily calculating it. We only exploit well-supportedness of $Post$ to lift it up to $Post'$. This is the methodology adopted in the case of `Vanilla`.

These intuitions are formalized in the following main theorem, stating that the General Criterion is a necessary and sufficient condition for declarative correctness of meta-interpreters.

Theorem 20 Suppose that $\vdash \{Pre\}\ P\ \{Post\}$ with $Post$ well-supported w.r.t. P and Pre. A program V is *declaratively correct* w.r.t. P and Pre iff it satisfies [GC] w.r.t. P and Pre.

Proof *If part.* As the hypotheses of [GC] hold, we can assume *(1-4)*. We now calculate, for $A \in Pre$:

$$
\begin{array}{ll}
 & A \in M_P^L \\
\Leftrightarrow & \quad \{ \text{ Theorem 11 } \} \\
 & A \in sp(P, Pre) \\
\Leftrightarrow & \quad \{ \text{ } Post \text{ well-supported and Theorem 14 } \} \\
 & A \in Post \cap Pre \\
\Leftrightarrow & \quad \{ \text{ [GC]}(3\text{-}4) \text{ } \} \\
 & \texttt{demo}(A) \in Post' \cap Pre' \\
\Leftrightarrow & \quad \{ \text{ Theorem 14 and [GC]}(1\text{-}2) \text{ } \} \\
 & \texttt{demo}(A) \in sp(V, Pre') \\
\Leftrightarrow & \quad \{ \text{ Theorem 11 and [GC] } (1) \text{ } \} \\
 & \texttt{demo}(A) \in M_V^L \cap Pre' \\
\Leftrightarrow & \quad \{ \text{ [GC]}(3) \text{ } \} \\
 & \texttt{demo}(A) \in M_V^L
\end{array}
$$

Only-if part. Let us consider $Pre' = B_L$ and $Post' = M_V^L$.

As for any program V the relation $\vdash \{B_L\}\ V\ \{B_L\}$ holds, by Theorems 11, 14 we have that $\vdash \{B_L\}\ V\ \{M_V^L\}$, i.e. *(1)*, and $M_V^L = sp(V, B_L)$ is well-supported, i.e. *(2)*. *(3)* is obvious.

To prove [GC]*(4)*, we calculate, for $A \in Pre$:

$$
\begin{array}{ll}
 & A \in Post \\
\Leftrightarrow & \quad \{ \text{ } Post \text{ well-supported and Theorems 11,14 } \} \\
 & A \in M_P^L \\
\Leftrightarrow & \quad \{ \text{ } V \text{ is declaratively correct } \} \\
 & \texttt{demo}(A) \in M_V^L \\
\Leftrightarrow & \quad \{ \text{ Definition of } Post' \text{ } \} \\
 & \texttt{demo}(A) \in Post'.
\end{array}
$$

□

By a simple weakening of the conditions *(1-4)*, a variant of the method is able to deal with *soundness*. Below, a Soundness Criterion [SC] is presented, together with a result showing that [SC] is a proof method for declarative soundness.

Definition 21 Suppose $\vdash \{Pre\}\ P\ \{Post\}$ with $Post$ well-supported w.r.t. P and Pre. A program V satisfies the Soundness Criterion [SC] with respect to P and Pre iff, for some Pre', $Post'$, the following properties hold:

1. $\vdash \{Pre'\}\ V\ \{Post'\}$,

2. $\forall A \in Pre.\quad \mathtt{demo}(A) \in Pre'$,

3. $\forall A \in Pre.\quad A \in Post \Leftarrow \mathtt{demo}(A) \in Post'$. $\square$

Theorem 22 Suppose that $\vdash \{Pre\}\ P\ \{Post\}$ with $Post$ well-supported w.r.t. P and Pre. A program V is *declaratively sound* w.r.t. P and Pre iff it satisfies [SC] w.r.t. P and Pre. $\square$

A further weakening allows us to characterize weak soundness. We conclude this section by defining a Weak Soundness Criterion [WSC] and stating its relation with weak soundness of meta-interpreters.

Definition 23 Suppose $\vdash \{Pre\}\ P\ \{Post\}$. A program V satisfies the Weak Soundness Criterion [WSC] with respect to P and $(Pre, Post)$ iff, for some Pre', $Post'$, the following properties hold:

1. $\vdash \{Pre'\}\ V\ \{Post'\}$,

2. $\forall A \in Pre.\quad \mathtt{demo}(A) \in Pre'$, *and*

3. $\forall A \in Pre.\quad A \in Post \Leftarrow \mathtt{demo}(A) \in Post'$. $\square$

Theorem 24 Suppose $\vdash \{Pre\}\ P\ \{Post\}$. A program V is *weak declaratively sound* w.r.t. P and $(Pre, Post)$ iff it satisfies [WSC] w.r.t. P and $(Pre, Post)$. $\square$

6.2 Amalgamation

Amalgamating an object program with its meta-interpreter means considering their union as a single program. This technique reveals to be a powerful tool for extending the logic programming paradigm with new features, provided by the intertwined use of the meta-program, which may offer new control facilities, and the object program, which may use these facilities. Assuming ambivalent syntax is a basic requirement for amalgamating object and meta-programs.

Simple amalgamation

Problems may arise if amalgamation is not used in a disciplined way. Consider, for instance, this program `Strange`:

```
p ← demo(q).
q.
```

and the following meta-interpreter V for `Strange`

```
demo(q).
```

It is simple to observe that V is declaratively correct w.r.t. `Strange` and B_L. However, amalgamating V and `Strange` leads to the following inconsistency:

$$V \cup \mathtt{Strange} \models \mathtt{p} \quad \text{and} \quad V \cup \mathtt{Strange} \not\models \mathtt{demo(p)}$$

Such a behaviour is *unsafe*, since $V \cup$ `Strange` should intuitively be declaratively correct with respect to itself. Next, we provide a formal account of this intuition, and a sufficient condition for *safe* amalgamation.

Definition 25 Suppose that V is declaratively correct w.r.t. P and Pre. We say that $V \cup P$ is a *safe amalgamation* for V and P, Pre iff for every $A \in Pre$

$$A \in M^L_{V \cup P} \quad \text{iff} \quad \texttt{demo}(A) \in M^L_{V \cup P}$$

□

An alternative way of reading this definition is to require that $V \cup P$ is declaratively correct w.r.t. $V \cup P$ and Pre, when V is declaratively correct w.r.t. P and Pre. In other words, declarative correctness lifts to the amalgamated meta-program.

The following theorem gives a simple sufficient condition for safe amalgamation. The underlying idea is to avoid interference between the object- and the meta- level. The proof is reported in the appendix.

Theorem 26 Consider a program P such that $\vdash \{Pre\}\ P\ \{Post\}$ with $Post$ well-supported w.r.t. P and Pre, and a program V which satisfies [GC] w.r.t. P and Pre. $V \cup P$ is a *safe amalgamation* for V and P, Pre if

1. $\vdash \{Pre \cup Pre'\}\ V \cup P\ \{Post \cup Post'\}$, and

2. $Post' \cap Pre \subseteq Post$, and

3. no atom in $Post$ has `demo` as predicate symbol.

where $Pre', Post'$ are the pre- and postconditions used in the proof of [GC]. □

As a result, we have that $Van(P) \cup P$ is a safe amalgamation for P and Pre, provided that no atom in P or in Pre has `demo` or `clause` as predicate symbol. Notice that the classical result on safety of the simple amalgamation of disjoint programs is a special case of Theorem 26.

Reflection up

A more liberal way of giving a meaning to the program `Strange`

```
p ←demo(q).
q.
```

is to interpret `demo(q)` as a call to the meta-level. This possibility is usually referred to as *reflection up* [19]. Let us consider the case of the `Vanilla`. To correctly implement the call to the meta-level, we need to add to `Vanilla` the clause

```
demo( demo(A) ) ←demo( A ).
```

We denote by $VU(P)$ the extended `Vanilla` instantiated by the object program P. Intuitively, by means of the added clause, a call to the predicate `demo` is interpreted at meta-level.

We formalize this intuition by appropriately defining $Pre_{VU(P)}$ and $Post_{VU(P)}$.

Definition 27 For a program P and $(Pre, Post)$, we define:

$$\begin{array}{rl} \texttt{demo}(B_1\ \texttt{\&}\ \dots\ \texttt{\&}\ B_n) \in Pre_{VU(P)} & \text{iff} \\ & \vdash \{Pre \cup Pre_{VU(P)}\}\ B_1, \dots, B_n\ \ \{Post \cup Post_{VU(P)}\} \\ \texttt{clause}(A,\ B) \in Pre_{VU(P)} & \text{iff} \quad true \end{array}$$

$$\begin{array}{rll} \texttt{demo}(B_1\ \texttt{\&}\ \dots\ \texttt{\&}\ B_n) \in Post_{VU(P)} & \text{iff} & Post \cup Post_{VU(P)} \models B_1, \dots, B_n \\ \texttt{clause}(A,\ B_1\ \texttt{\&}\ \dots\ \texttt{\&}\ B_n) \in Post_{VU(P)} & \text{iff} & A \leftarrow B_1, \dots, B_n\ \in ground_L(P) \end{array}$$

□

Next theorem expresses that $VU(P)$ is a declaratively correct meta-interpreter w.r.t. $VU(P) \cup P$ and Pre, formalizing the intuitive notion of that a call to the `demo`-predicate is interpreted at meta-level.

Theorem 28 Suppose that $\vdash\ \{Pre \cup Pre_{VU(P)}\}\ P\ \{Post \cup Post_{VU(P)}\}$ holds with $Post \cup Post_{VU(P)}$ well-supported w.r.t. $P \cup VU(P)$ and $Pre \cup Pre_{VU(P)}$. If no head of a clause from P and no atom in Pre or $Post$ has `demo` or `clause` as predicate symbol, then $VU(P)$ is declaratively correct w.r.t. $P \cup VU(P)$ and Pre. □

A proof of this fact can be found in the appendix. As an example, consider again `Strange`, and $Pre = Post = \{\ \texttt{p}, \texttt{q}\ \}$. It is easy to check that we are in the hypothesis of Theorem 28, and then to conclude that

$$\texttt{p} \in M^L_{\texttt{Strange} \cup VU(\texttt{Strange})} \quad \text{iff} \quad \texttt{demo(p)} \in M^L_{VU(\texttt{Strange})}$$

and analogously for `q`.

Reflection down

We conclude the overview of the amalgamation techniques, looking at reflection down, i.e. the possibility of calling the object-level from the meta-level.
The simplest reflective meta-interpreter is the standard Prolog implementation of the predicate `call` by means of *meta-variables* [31]. For a program P we denote by $Call(P)$, the meta-program consisting of P together with the clause

```
demo(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 a meta-variable is selected.

To properly tackle the problem we assume the use of the *terms-as-atoms* ambivalent syntax [22]. In our context of ambivalent syntax, we simply assume $\Sigma_L = \Pi_L$, 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 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.
Let us consider a program P such that $\vdash \{Pre\}\ P\ \{Post\}$, and suppose that no atom in Pre or $Post$ or P has `demo` as a predicate symbol. We will show that $Call(P)$ satisfies [GC] w.r.t. P and Pre.
First, we point out that

$$\vdash \{Pre_{Call(P)}\}\ Call(P)\ \{Post_{Call(P)}\}$$

where

$$\begin{aligned} Pre_{Call(P)} &= Pre \cup \{\ \mathtt{demo}(A) \mid A \in Pre\ \} \\ Post_{Call(P)} &= Post \cup \{\ \mathtt{demo}(A) \mid A \in Post\ \} \end{aligned}$$

Actually, the proof obligations for the clauses from P are trivially satisfied, since $\vdash \{Pre\}\ P\ \{Post\}$ holds, and

$$rel(A) \neq \mathtt{demo} \wedge A \in Pre_{Call(P)} \Leftrightarrow A \in Pre$$

and analogously for $Post_{Call(P)}$.
The proof outline for the clause of `demo` is straightforward.

$\{\ X \in Pre\ \}$
`demo( X )` $\leftarrow$
$\quad\{\ X \in Pre\ \}$
`X.`
$\quad\{\ X \in Post\ \}$
$\{\ X \in Post\ \}$

This concludes the proof of [GC] *(1)*. Let us show that [GC] *(2)* holds.
Assuming $Post$ well-supported, we show that $Post_{Call(P)}$ is well-supported as well. By hypothesis, there exists a well-founded poset $(W, <)$ [1] 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

$$\forall i \in [1, n] : Post \models B_i \wedge |A| > |B_i| \tag{7}$$

Consider now any function $||\ ||_c : B_L \rightarrow W$ such that, for $A \in Pre$

$$\begin{gathered} ||\mathtt{demo(A)}||_c > |A| \\ ||A||_c = |A| \end{gathered}$$

We claim that there exist proof outlines for the clauses of $Call(P)$ and $||\ ||_c$. This is immediate for the clause from P, by hypothesis. The proof outline for the clause of `demo` is

$\{\ X \in Pre \cap Post\ \}$	
`demo( X )` $\leftarrow$	$\{\ ||\mathtt{demo}(X)||_c\ \}$
`X.`	$\{\ |X|\ \}$
$\{\ X \in Post\ \}$	

[1] Without lack of generality, we can assume it without maximal elements.

Therefore, we proved [GC] *(2)*. [GC] *(3-4)* hold by definition of $Pre_{Call(P)}$ and $Post_{Call(P)}$.

We summarize this correctness result in the following theorem. In addition, we point out that the hypothesis of the theorem are sufficient to state the absence of run-time errors due to meta-variable selection.

Theorem 29 Suppose $\vdash \{Pre\}\ P\ \{Post\}$, and that no atom in *Pre* or *Post* or P has `demo` as a predicate symbol. Then $Call(P)$ is declaratively correct w.r.t. P and *Pre*. Moreover, for any query Q such that $\vdash \{Pre_{Call(P)}\}\ Q\ \{Post_{Call(P)}\}$, no meta-variable is selected in a LD-derivation for $Call(P)$ and Q.

Proof We proved that $Call(P)$ satisfy [GC] w.r.t. P and *Pre*. Then, by Theorem 20, it is a declaratively correct meta-interpreter for P.

Suppose now that a meta-variable X is selected in a LD-derivation for $Call(P)$ and Q. Then, by Lemma 5 (i), we have

$$Pre_{Call(P)} \models X$$

and, since $\Pi_L \subseteq \Sigma_L$, $\quad Pre_{Call(P)} = B_L$. By definition of $Pre_{Call(P)}$, this implies that `demo(`t`)` is in *Pre* for some t. This contradicts the hypothesis that no atom with `demo` as predicate symbol belongs to *Pre*. □

6.3 Extensions

Several other extensions of `Vanilla` are reported in the literature (see [31].) The approach based on the general criterion [GC] can be applied systematically. However, we need to extend the definition of declarative correctness and [GC] to consider meta-predicates `demo` with more than one argument. A simple way to do that, is to consider a top level predicate `demo` defined as follows:

$$\texttt{demo2(A)} \leftarrow \texttt{demo(A, }\mathbf{B}\texttt{).}$$

where $\mathbf{B}$ is the collection of the additional arguments of `demo`. In this case, however, the declarative correctness states a *weak* relation between A and the other arguments, namely that $A \in M_P^L$ iff there exists $\mathbf{B} \in U_L$ such that $\texttt{demo(A, }\mathbf{B}\texttt{)} \in M_V^L$.

A stronger property is obtained as follows. We denote with $\mathcal{R}$ an arbitrary assertion.

Definition 30 A program V is $\mathcal{R}$*-declaratively correct* with respect to a program P and a set $Pre \subseteq B_L$ iff for every $A \in Pre$ and $\mathbf{B} \in U_L$

$$\mathcal{R}(A, M_P^L \cap Pre, \mathbf{B}) \quad \text{iff} \quad \texttt{demo(}A\texttt{, }\mathbf{B}\texttt{)} \in M_V^L$$

□

Definition 1 is a special case of Definition 30, obtained considering $\mathcal{R}(A, M_P^L \cap Pre, \mathbf{B}) \overset{def}{\Leftrightarrow} A \in M_P^L \cap Pre$. By a simple change of conditions *(3,4)* in Definition 19, Theorem 20 is extended, i.e. [GC] with the new conditions *(3',4')* is a necessary and sufficient criterion for proving $\mathcal{R}$-declarative correctness.

(3') $\forall\, \mathbf{B} \in U_L\ \forall A \in Pre.\ \ \texttt{demo(}A\texttt{, }\mathbf{B}\texttt{)} \in Pre'$,
(4') $\forall\, \mathbf{B} \in U_L\ \forall A \in Pre.\ \ \mathcal{R}(A, Post \cap Pre, \mathbf{B}) \Leftrightarrow \texttt{demo(}A\texttt{, }\mathbf{B}\texttt{)} \in Post'$.

In this rest of this section, we briefly sketch the correctness proof of one extension of `Vanilla`, namely the `Proof-Tree Builder Vanilla` [31]. For a program P, we denote with $PTB(P)$ the program

```
demo( empty, true ).
demo( X & Y, TX & TY ) ←
   demo( X, TX ),
   demo( Y, TY ).
demo( X, X if T) ←
   clause(X, Y),
   demo(Y, T).
clause( A , B₁ & ... & Bₙ ).          for every A ←B₁, ..., Bₙ ∈ P
```

This meta-interpreter is devised for yielding a *justification* for any logical consequence of the object program. First of all, we give a formal definition of proof trees (see [18].)

Definition 31 The following rules define the proof trees for ground queries, and a program P:

1. T_1 `&` $\ldots$`&` T_n is a proof tree for $B_1, \ldots, B_n$, if for $i \in [1, n] (n \geq 0)$, T_i is a proof tree for B_i,
2. A `if` T is a proof tree for A, if T is a proof tree for the query $B_1, \ldots, B_n$, and $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$. □

Notice that the base case is *(1)* for $n = 0$, i.e. `true` is a proof tree for the empty query.

In [18] it is showed that $A \in M_P^L$ iff there exists a proof tree for A. In this sense, a proof tree is a justification for A.

After these preliminaries, we construct suitable pre- and postconditions for $PTB(P)$ starting from those for P:

$$\begin{array}{rcl} \texttt{demo(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{, }T\texttt{)} \in Pre_{PTB(P)} & \text{iff} & \vdash \{Pre\}\ B_1, \ldots, B_n\ \{Post\} \\ \texttt{clause(}A\texttt{, }B\texttt{)} \in Pre_{PTB(P)} & \text{iff} & true \end{array}$$

$$\begin{array}{rcl} \texttt{demo(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{, }T\texttt{)} \in Post_{PTB(P)} & \text{iff} & Post \models B_1, \ldots, B_n\ \wedge \\ & & T \text{ is a proof tree for } B_1, \ldots, B_n \\ \texttt{clause(}A\texttt{, }B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{)} \in Post_{PTB(P)} & \text{iff} & A \leftarrow B_1, \ldots, B_n \in ground_L(P) \end{array}$$

It is a simple exercise to prove that $PTB(P)$ satisfies the general criterion [GC] w.r.t. P and Pre if $\vdash \{Pre\}\ P\ \{Post\}$ and $Post$ well-supported. Therefore, the correctness of this meta-interpreter follows from the extension of Theorem 20 previously mentioned, where $\mathcal{R}(A, I, \mathbf{B}) \overset{def}{\Leftrightarrow} I \models A \wedge T$ is a proof tree for A.

Theorem 32 Suppose that $\vdash \{Pre\}\ P\ \{Post\}$. Then for $A \in Pre$ and every ground term T

$$T \text{ is a proof tree for } A \quad \text{iff} \quad \texttt{demo(}A\texttt{,}T\texttt{)} \in M^{L}_{PTB(P)}$$

□

6.4 Parameterization: modular logic programming

Several extensions of `Vanilla` were proposed in the past, in order to enhance the expressiveness of the logic programming paradigm by adding new features. One of the most studied problems concerns modular programming. We refer the reader to [17, 13, 14] for an overview of the topic.

In this section, we will study a parameterization of the `Vanilla` extended to deal with modular programming by means of reflection up. In the meta-program, the `demo` relation has an extra predicate which is used to refer to a hierarchy of modules, i.e., separate programs. A set of programs $\{P_i\}_{i\in\{1,n\}}$ is a module hierarchy iff for $i \in [1, n]$ if $\texttt{demo(}X\texttt{, }A\texttt{)}$ appears in the body of a clause from P_i then $X = \texttt{p}_j \wedge j < i$, where each $\texttt{p}_i$ is a distinct constant denoting a module. Intuitively, in a module hierarchy, each module can use, besides its own relations, only relations defined in lower modules.

For a module hierarchy $\{P_i\}_{i\in\{1,n\}}$, we write $P_1 \lhd \ldots \lhd P_n$, and define $VanR(\lhd_{i=1}^{n} P_i)$ the program

```
demo( X, true ).
demo( X, A & B ) ←
   demo( X, A ),
   demo( X, B ).
demo( X, demo(Y, A) ) ←
   demo(Y , A).
demo( X, A ) ←
   clause( X, A, B ),
   demo( X, B ).
clause( pᵢ, A , B₁ & ... & Bₙ ).
```

for every $\texttt{A} \leftarrow \texttt{B}_1\texttt{, ..., B}_n \in P_i$, and $i \in [1, n]$

As an example, consider the following module

```
Module a

gt( X, Z ) ←
   demo(b , generate(X,Z) ),
   demo(c, test(Z) ).
```

This is a *generate & test* schema, where the generation procedure is defined in module `b` and the test in module `c`.

Intuitively, a *stratification* of meta-calls is introduced, in the sense that meta-calls are allowed on lower modules only. This is a simple notion of modular logic programming, but as long as we deal with least Herbrand model semantics no full treatment of the union composition operator can be performed. The reader can find an extensive treatment of composition operators and of related fully abstract semantics in [17].

As usual, the crucial step in dealing with correctness of `Vanilla` with modules is the definition of pre and postconditions. The definition is given by induction on the length of the module hierarchy.

Definition 33 We define $Pre_{VanR(\lhd_{i=1}^{n} P_i)}$ and $Post_{VanR(\lhd_{i=1}^{n} P_i)}$ starting from $\{Pre_{P_i}\}_{i \in \{1,n\}}$ and $\{Post_{P_i}\}_{i \in \{1,n\}}$ for $i \in [1, n]$. For $n = 0$ both are the empty set.
Suppose now $n > 0$ and $Q = P_1 \lhd \ldots \lhd P_{n-1}$, $P = P_n$, and $\mathtt{p} = \mathtt{p}_n$.
For $Q \lhd P$ we define:

$$\begin{array}{rl}
\mathtt{demo(}x\mathtt{,}B_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ B_k\mathtt{)} \in Pre_{VanR(Q \lhd P)} & \text{if} \\
 & \mathtt{demo(}x\mathtt{,}B_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ B_k\mathtt{)} \in Pre_{VanR(Q)} \\
\mathtt{clause(}x\mathtt{,}\ A\mathtt{,}\ B\mathtt{)} \in Pre_{VanR(Q \lhd P)} & \text{iff} \\
 & \mathtt{clause(}x\mathtt{,}\ A\mathtt{,}\ B\mathtt{)} \in Pre_{VanR(Q)} \\
\mathtt{demo(p,}\ B_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ B_k\mathtt{)} \in Pre_{VanR(Q \lhd P)} & \text{if} \\
 & \vdash \{Pre_P \cup Pre_{VanR(Q)}\}\ B_1, \ldots, B_k\ \{Post_P \cup Post_{VanR(Q)}\} \\
\mathtt{clause(p,}\ A\mathtt{,}\ B\mathtt{)} \in Pre_{VanR(Q \lhd P)} & \text{iff}\quad true
\end{array}$$

$$\begin{array}{rl}
\mathtt{demo(}x\mathtt{,}B_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ B_k\mathtt{)} \in Post_{VanR(Q \lhd P)} & \text{if} \\
 & \mathtt{demo(}x\mathtt{,}B_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ B_k\mathtt{)} \in Post_{VanR(Q)} \\
\mathtt{clause(}x\mathtt{,}\ A\mathtt{,}\ B_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ B_k\mathtt{)} \in Post_{VanR(Q \lhd P)} & \text{if} \\
 & \mathtt{clause(}x\mathtt{,}\ A\mathtt{,}\ B_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ B_k\mathtt{)} \in Post_{VanR(Q)} \\
\mathtt{demo(p,}B_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ B_k\mathtt{)} \in Post_{VanR(Q \lhd P)} & \text{if} \\
 & Post_P \cup Post_{VanR(Q)} \models B_1, \ldots, B_k \\
\mathtt{clause(p,}\ A\mathtt{,}\ B_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ B_k\mathtt{)} \in Post_{VanR(Q \lhd P)} & \text{if} \\
 & A \leftarrow B_1, \ldots, B_k\ \in ground_L(P)
\end{array}$$

□

Intuitively, at level i, for $i \in [1, n]$,

- the specification of the lower levels ($< i$) is known, and
- the specification of the module i can be defined starting from those of lower modules, and from the specification $(Pre, Post)$, for the symbols of P_i.

Next, we need a notion of correctness for `Vanilla` with modules, which can be obtained by extending Definition 1 in a natural way.

Definition 34 A program V is *declaratively correct* with respect to a module hierarchy $\{P_i\}_{i \in \{1,n\}}$ and a set $\{Pre_{P_i}\}_{i \in \{1,n\}}$ iff for every $i \in [1, n]$, if $A \in Pre_{P_i}$ then

$$A \in M^L_{P_i \cup V} \quad \text{iff} \quad \mathtt{demo(p}_i\mathtt{,}\ A\mathtt{)} \in M^L_V$$

□

The intuition is that the meta-program V is required to be a declaratively correct meta-interpreter for $P_i \cup V$, for each level $i \in [1, n]$ of the module hierarchy. It should be observed that this is a form of amalgamation. An atom `demo(p`$_j$`,`A`)` in the body of a clause from P_i is interpreted as a call to a lower module P_j.

The following theorem expresses the correctness of the `Vanilla` extended with modules.

Theorem 35 Let us consider a module hierarchy $\{P_i\}_{i\in\{1,n\}}$, and two sets of interpretations $\{Pre_{P_i}\}_{i\in\{1,n\}}$ and $\{Post_{P_i}\}_{i\in\{1,n\}}$. Assume that no atom with `demo` or `clause` as predicate symbol appears in Pre_{P_i} or $Post_{P_i}$ or in the head of a clause from P_i, for $i \in [1, n]$.

Let, for $i \in [1, n+1]$ $\quad Q_i = P_1 \lhd \ldots \lhd P_{i-1}$.

Assume that, for $i \in [1, n-1]$

$$\vdash \{Pre_{VanR(Q_i)} \cup Pre_{P_i}\}\ P_i\ \{Post_{VanR(Q_i)} \cup Post_{P_i}\} \tag{8}$$

$$\text{with } Post_{VanR(Q_i)} \cup Post_{P_i} \text{ well-supported} \tag{9}$$

$$\text{w.r.t. } P_i \cup VanR(Q_i) \text{ and } Pre_{VanR(Q_i)} \cup Pre_{P_i}$$

and

$$\vdash \{Pre_{VanR(Q_n)} \cup Pre_{P_n}\}\ P_n\ \{Post_{VanR(Q_n)} \cup Post_{P_n}\} \tag{10}$$

Then $VanR(\lhd_{i=1}^{n} P_i)$ is declaratively correct w.r.t. $\{P_i\}_{i\in\{1,n\}}$ and $\{Pre_{P_i}\}_{i\in\{1,n\}}$. $\square$

Observe that in the above theorem, the condition (10) on the top module P_n, which is weaker than conditions (8), (9), is sufficient, as it is not necessary that the considered postcondition is well-supported for P_n.

7 Total Correctness

The relation $\vdash$ is not sufficient for dealing with (weak) total correctness, as termination is not taken into account. Relation $\vdash_t$ deals with termination by means of level mappings, which decrease from the head to the body atoms of a ground instance of a clause (see Definition 3.)
In addition to the properties of $\vdash$, the relation $\vdash_t$ allows to prove *universal termination* w.r.t. LD-resolution, i.e. SLD-resolution with the leftmost selection rule. The next definition is an adaptation the notion of terminating programs used in the logic programming termination literature.

Definition 36 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. $\square$

Next Theorem shows that, if $\vdash \{Pre\}\ P\ \{Post\}$ holds then P is Pre-terminating iff $\vdash_t \{Pre\}\ P\ \{sp(P, Pre)\}$ holds (for a proof, see [29, 28].) In other words, the relation $\vdash_t$ is a complete termination proof method relatively to the triples in $\vdash$.

Theorem 37 If $\vdash_t \{Pre\}\ P\ \{Post\}$ holds then P is Pre-terminating. Conversely, $\vdash \{Pre\}\ P\ \{Post\}$ and P Pre-terminating imply $\vdash_t \{Pre\}\ P\ \{sp(P, Pre)\}$. $\square$

As an immediate corollary of this termination property, we have the Total Correctness Theorems.

Theorem 38 ((Weak) Total Correctness) If $\vdash_t \{Pre\}\ P\ \{Post\}$ holds, then P is weak totally correct w.r.t. the specification $(Pre, Post)$, and totally correct w.r.t. the specification $(Pre, sp(P, Pre))$. $\square$

The next step is to find a proof method for total correctness. A straightforward consequence of the results of Section 5.2 serves our purposes.

Theorem 39 Let P be a program such that $\vdash_t \{Pre\}\ P\ \{Post\}$. Then the following statements are equivalent:

(i) $Post$ is well-supported w.r.t. P and Pre,

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

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

An immediate application of this theorem is a simpler reformulation of the proof outline for proving that a given postcondition is well-supported. The well-founded poset $(W, <)$, the function $|\ | : B_L \rightarrow W$ and the proof obligation (iii) disappear (actually, these requirements are already satisfied for $W = N$ and the level mapping used to prove $\vdash_t \{Pre\}\ P\ \{Post\}$.) As a consequence, this also simplifies proving the requirements of [GC].

Proof Outlines and Vanilla

We now show that also the relation $\vdash_t$ lifts up from the object program to the `Vanilla`, i.e.

$$\vdash_t \{Pre_{Van(P)}\}\ Van(P)\ \{Post_{Van(P)}\} \tag{11}$$

for some $|\ |_{Van(P)}$ when $\vdash_t \{Pre\}\ P\ \{Post\}$ holds.

Let us assume that $|\ |$ is the level mapping used to prove $\vdash_t \{Pre\}\ P\ \{Post\}$, and recall the function $||\ ||$

$$||\texttt{demo(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{)}|| = bag(|B_1|, \ldots, |B_i|)$$

if $Post \models B_1, \ldots, B_{i-1}$ and $Post \not\models B_i$. It is a simple exercise to prove

$$\vdash_t \{Pre_{Van(P)}\}\ Van(P)\ \{Post_{Van(P)}\}$$

using the level mapping $|\ |_{Van(P)}$ such that for $A \in B_L$

$$|A|_{Van(P)} = \mu(||A||)$$

where $\mu : bag(N) \rightarrow N$ is a *partially monotonic function* [2].

For instance the proof outlines for the clauses *(b,c)* are

$$
\begin{array}{lll}
 & \{\ \vdash \{Pre\}\ A, B^-\ \{Post\}\ \} & \\
(b) & \texttt{demo(A \& B)} \ \leftarrow & \{\ \mu(\|\texttt{demo(}A\texttt{ \& }B\texttt{))}\|)\ \} \\
 & \quad\quad \{\ Pre \models A\ \} & \\
 & \quad \texttt{demo(A),} & \{\ \mu(\|\texttt{demo(}A\texttt{)}\|)\ \} \\
 & \quad\quad \{\ Post \models A\ \} & \\
 & \quad\quad \{\ \vdash \{Pre\}\ B^-\ \{Post\}\ \} & \\
 & \quad \texttt{demo(B).} & \{\ \mu(\|\texttt{demo(}B\texttt{)}\|)\ \} \\
 & \quad\quad \{\ Post \models B^-\ \} & \\
 & \{\ Post \models A, B^-\ \} &
\end{array}
$$

$$
\begin{array}{lll}
 & \{\ \vdash \{Pre\}\ A\ \{Post\}\ \} & \\
(c) & \texttt{demo(A)} \ \leftarrow & \{\ \mu(\|\texttt{demo(}A\texttt{))}\|)\ \} \\
 & \quad\quad \{\ true\ \} & \\
 & \quad \texttt{clause(A, B),} & \{\ \mu(bag())\ \} \\
 & \quad\quad \{\ A \leftarrow B^- \in ground_L(P)\ \ )\ \} & \\
 & \quad\quad \{\ \vdash \{Pre\}\ B^-\ \{Post\}\ \} & \\
 & \quad \texttt{demo(B).} & \{\ \mu(\|\texttt{demo(}B\texttt{)}\|)\ \} \\
 & \quad\quad \{\ Post \models B^-\ \} & \\
 & \{\ Post \models A\ \} &
\end{array}
$$

As an example, the decreasing of the level mapping in proof outline *(c)* is given by the following property

$$\mu(\|\texttt{demo(}B\texttt{)}\|) < \mu(\|\texttt{demo(}A\texttt{)}\|)$$

for $A \leftarrow B^- \in ground_L(P)$), which holds by the hypothesis on μ.

Moreover, as in case of relation $\vdash$, we notice that for a (not necessarily ground) query $B_1, \ldots, B_n$

$$\vdash_t \{Pre_{Van(P)}\}\ \texttt{demo(}B_1\texttt{ \& } \ldots \texttt{ \& }B_n\texttt{)}\ \{Post_{Van(P)}\}$$

when $\vdash_t \{Pre\}\ B_1, \ldots, B_n\ \ \{Post\}$.

The next corollary shows that termination lifts up from the object program to the `Vanilla`.

Corollary 40 If $\vdash \{Pre\}\ P\ \{Post\}$ holds and P is *Pre*-terminating then $Van(P)$ is $Pre_{Van(P)}$-terminating.

Proof If P is *Pre*-terminating then by Theorem 37 $\vdash_t \{Pre\}\ P\ \{sp(P, Pre)\}$. From (11) and Theorem 37 again, we have that $Van(P)$ is $Pre_{Van(P)}$-terminating. □

[2] A monotonic function from $bag(N)$ to N does not exist, as $bag(N)$ is not an ω-ordering. However, we need a weaker property. Let r be the maximum number of atoms in the body of a clause from P. μ is partially monotonic if $\mu(b_1) > \mu(b_2)$ for the bag b_2 obtained replacing an element of b_1 by at most r elements lower than it. It is possible to show that it is a well-formed definition and that such a function exists.

An analogous result for arbitrary meta-interpreters can be achieved by rephrasing [GC] with the $\vdash_t$ relation.

In conclusion, we observe that `Vanilla` is *closed w.r.t. the proof theory notions* that have been introduced in the paper, in the sense that (weak) partial and total correctness systematically lifts up from the object to the meta-level, in case of vanilla, its enhancements, and, more generally, of any meta-program which satisfies the general criterion. In all these cases we can directly derive the declarative correctness of the considered interpreter.

8 Language independence

8.1 Relations $\vdash$ and $\vdash_t$ are language independent

So far, we have assumed that L is the underlying language of any program. This fact is very useful in modular proofs of correctness [2, 27] and amalgamation of object and meta-program. The alternative solution of considering L_P, the language generated by P, does not permit to reuse the proofs of correctness in a modular environment (for instance, when we want to prove correctness of the union of two programs which have been separately verified.)

However, we report a theorem from [28] showing that the relations $\vdash$ and $\vdash_t$ are essentially language independent. More precisely, 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. By the next Theorem, the relation $\vdash$ can be conservatively extended to any language extending L_P.

Theorem 41 Consider $\vdash \{Pre\}\ P\ \{Post\}$ defined on L. Let L' be a language extending L. Then $\vdash \{Pre^u\}\ P\ \{Post^u\}$ where

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

$\square$

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 $\vdash_t$.

It is worth noting that Theorem 41 allows us to prove correctness of meta-interpreters in a modular way. For example, consider a language L, and prove that `Vanilla` satisfies the [GC] with respect to P and Pre. If sometimes after we need to deal with an extension L' of L, Theorems 41 and 16 imply that the `Vanilla` is still declaratively correct with respect to P and Pre (since $Pre \subseteq Pre^u$.)

8.2 Language independent programs

Another way of tackling the problem of reasoning on different languages was devised in Martens and De Schreye [26], which restricted the attention to the class of language independent logic programs.

Definition 42 A program P is *language independent* if for every extension L of L_P, $M_P^L = M_P$. $\square$

Martens and De Schreye proved that for a language independent program P the following soundness result holds:
[Sli] For $A \in B_{\langle \Sigma_{Van(P)}, \Pi_P \rangle}$, if $\texttt{demo}(A) \in M_{Van(P)}$, then $A \in M_P$.

[Sli] essentially states declarative soundness for all atoms with object-level relations and meta-level functors. Kalsbeek [23] proved that [Sli] is also a sufficient condition to show that P is language independent. However, the proof is rather complicated and uses results from Hilbert calculi.
In the following, a simpler and direct proof is given, exploiting the correctness results and the following well-known fact:

A program P is language independent iff $M_P = M_P^L$ for $L = \langle \Sigma_{Van(P)}, \Pi_P \rangle$.

Theorem 43 If P satisfies [Sli] iff it is language independent.

Proof *Only if part.* Using the fact above, we only have to show that $M_P = M_P^L$, for $L = \langle \Sigma_{Van(P)}, \Pi_P \rangle$. Clearly $M_P \subseteq M_P^L$. Let us prove the other inclusion. Consider $A \in M_P^L$. By Theorem 16 with $Pre = B_L$, $\texttt{demo}(A) \in M_{Van(P)}$. As [Sli] holds, we conclude that $A \in M_P$.

If part. Let A be in $B_{\langle \Sigma_{Van(P)}, \Pi_P \rangle}$. If $\texttt{demo}(A) \in M_{Van(P)}$, then by correctness of `Vanilla` (Theorem 16) $A \in M_P^L$, for a language L rich enough. By Definition 42, we conclude $A \in M_P$. $\square$

9 Correct and computed instances of queries

So far, we limited our attention to relations between ground consequences of the object-level and meta-level programs, i.e., to ground correct instances of queries. In this section we show how it is possible to study relations between non-ground correct and computed instances of object- and meta-level programs. Once again, our reasoning is based on relation $\vdash$.

9.1 Least term model semantics

We now study conditions under which the meta-program preserves the (non-ground) correct one-atom queries of the object program, also known as the least term model semantics or $\mathcal{C}$-semantics (see [5].)

Definition 44 A program V is *$\mathcal{C}$-correct* with respect to a program P and $Pre \subseteq B_L$ iff for every $A \in Atom_L$ such that $Pre \models A$

$$P \models A \quad \text{iff} \quad V \models \texttt{demo}(A)$$

□

The following lemma is crucial to the purpose of verifying $\mathcal{C}$*-correctness*. A proof is reported in the appendix.

Lemma 45 A program V is $\mathcal{C}$*-correct* with respect to a program P and $Pre \subseteq B_L$ iff it is *declaratively correct* with respect to P and $Pre^u \subseteq B_{L'}$, where

- L' is a language obtained adding to L infinitely many new constant symbols, and
- $Pre^u = \bigcup_{B \in Atom_L \ \wedge \ Pre \models B} [B]_{L'}$. □

Notice that Pre^u is defined on the language L' by considering ground instances on L' of the atoms of L which are true in Pre. Intuitively, $\mathcal{C}$-correctness can be rephrased in terms of declarative correctness, simply considering an extended language.

Theorem 46 Consider a program P such that $\vdash \{Pre\}\ P\ \{Post\}$, with $Post$ well-supported w.r.t. P and Pre.
A program V is $\mathcal{C}$*-correct* with respect to P and a Pre iff it is satisfies [GC] with respect to P (defined on L') and $Pre^u \subseteq B_{L'}$, where

- L' is a language obtained adding to L infinitely many new constant symbols, and
- $Pre^u = \bigcup_{B \in Atom_L \ \wedge \ Pre \models B} [B]_{L'}$.

Proof By Theorem 41, we have $\vdash \{Pre^u\}\ P\ \{Post^u\}$. By Theorem 20, V is declaratively correct w.r.t. P and Pre^u iff it satisfies [GC] with respect to P (defined on L') and Pre^u. By Lemma 45, this is equivalent to show that V is $\mathcal{C}$*-correct* with respect to P and a Pre. □

As a result we conclude the $\mathcal{C}$-correctness of the `Vanilla`.

Theorem 47 Suppose that $\vdash \{Pre\}\ P\ \{Post\}$. Then $Van(P)$ is $\mathcal{C}$*-correct* with respect to P and Pre.

Proof By Theorem 16, $Van(P)$ is declaratively correct with respect to P and Pre^u, as $\vdash \{Pre^u\}\ P\ \{Post^u\}$ holds for $Pre^u, Post^u$ defined as in Theorem 46. The conclusion then follows from Lemma 45 and Theorem 16. □

9.2 Computed instances of queries

Recently, Apt et al. introduced a method for characterizing the computed instances of queries on the basis of the declarative semantics [5]. The method is developed with reference to the notion of partial correctness introduced in this paper, namely $\vdash \{Pre\}\ P\ \{Post\}$. 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 48 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 [5], 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 49 Assume that the signature of the underlying language L contains infinitely many function symbols, i.e. Σ_L is infinite. 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 computed instances of Q are the most general instances of Q which are true in $Post$. □

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 48 directly is not handy, as the notion of redundancy-freedom is essentially procedural. To solve this problem, several sufficient conditions are studied in [5] based either on syntactic decidable properties or on declarative semantics, which allow us to prove that a given program is $(Pre,\ Post)$-redundancy-free.

We refer here to the following conditions SYN and SEM adapted from [5], which, for a program P such that $\vdash \{Pre\}\ P\ \{Post\}$ with $Post$ well-supported w.r.t. P and Pre, imply that P is $(Pre,\ Post)$-redundancy-free.

SYN. No variable disjoint variants of any two distinct clauses of P unify.
SEM. If $H \leftarrow \mathbf{B}_1$ and $H \leftarrow \mathbf{B}_2$ are distinct ground instances from $ground(P)$ then

$$Post \not\models H \wedge \mathbf{B}_1 \wedge \mathbf{B}_2.$$

As it shown in the [5], the conditions SYN and SEM can be applied to a large class of natural Prolog programs—for these programs we are able to characterize their output on the basis of purely declarative reasoning. It is therefore legitimate to ask whether the conditions SYN and SEM lift from the object program up to the meta-interpreter. The following simple observation shows that this is indeed the case.

Theorem 50 Consider a program P such that $\vdash \{Pre\}\ P\ \{Post\}$, and assume that P satisfies conditions SYN and SEM. Then $Van(P)$ also satisfies conditions SYN and SEM.

Proof By the assumption that $\vdash \{Pre\}\ P\ \{Post\}$ holds, $\vdash \{Pre_{Van(P)}\}\ Van(P)$ $\{Post_{Van(P)}\}$ holds as well. Also, for any ground query Q^-, we have that:

$$Post \models Q^- \text{ iff } Post_{Van(P)} \models \texttt{demo}(Q). \tag{12}$$

To prove that $Van(P)$ satisfies SYN we reason as follows. Consider a pair of disjoint variants of clauses from $Van(P)$. If the two clauses define the `demo` predicate, then they clearly do not unify. If the two clauses define the `clause` predicate, then they unify iff the object program P violates SYN. This concludes the proof that $Van(P)$ satisfies SYN.

To prove that $Van(P)$ satisfies SEM we reason as follows. The only non trivial case arises considering clause $(d3)$. Assume by contradiction that for two distinct instances of $(d3)$, say $\texttt{demo(}A\texttt{)} \leftarrow \texttt{clause(}A\texttt{,}B\texttt{), demo(}B\texttt{)}$ and $\texttt{demo(}A\texttt{)} \leftarrow \texttt{clause(}A\texttt{,}B'\texttt{), demo(}B'\texttt{)}$ we have that $Post_{Van(P)} \models \texttt{demo(}A\texttt{)}, \texttt{clause(}A\texttt{,}B\texttt{)}, \texttt{demo(}B\texttt{)}, \texttt{clause(}A\texttt{,}B'\texttt{)}, \texttt{demo(}B'\texttt{)}$. By the definition of $Post_{Van(P)}$ we have that $A \leftarrow B^-$ and $A \leftarrow B'^-$ are two distinct clauses from $ground(P)$. By (12) we have $Post \models A \wedge B^- \wedge B'^-$, which implies that the object program P violates SEM. This concludes the proof that $Van(P)$ satisfies SEM. □

As a consequence of this result, under the hypothesis of Theorem 49 we can conclude that the computed instances of P and any query Q^- such that $\vdash \{Pre\}\ Q^-\ \{Post\}$ coincide with the computed instances of $Van(P)$ and $\texttt{demo(}Q\texttt{)}$. Moreover, those computed instances can be retrieved from the declarative semantics, in the sense that the following are equivalent:

1. Q_1^- is a computed instance of P and Q^-;
2. $\texttt{demo(}Q_1\texttt{)}$ is a computed instance of $Van(P)$ and $\texttt{demo(}Q\texttt{)}$;
3. Q_1^- is a most general instance of Q^- which is true in $Post$;
4. $\texttt{demo(}Q_1\texttt{)}$ is a most general instance of $\texttt{demo(}Q\texttt{)}$ which is true in $Post_{Van(P)}$.

It should be noted however that other conditions from [5], such as SEM1 and SEM2, do not immediately lift up to the Vanilla meta-interpreter.

10 Conclusions

In this paper, we applied the proof principles from the verification method of [28, 29] to some well-known meta-programs from the logic programming folklore, such as the `Vanilla` meta-interpreter, the simple reflective meta-interpreter, and some extensions and parameterizations, including forms of amalgamation. As a by-product of this study, we have extended the standard notions of *declarative correctness* and *soundness* and designed a criterion [GC] for declarative correctness of general meta-interpreters. The mentioned results hold in the presence of *ambivalent syntax* as well.

The contribution of this 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,
- correct instances of queries,
- computed instances of queries.

Interestingly, it is possible to prove all this on the basis of purely declarative reasoning, using the mentioned proof method.

Concerning the non-ground case, a criterion for $\mathcal{C}$-correctness was given, and a large class (Turing equivalent) of programs is devised for which we can reason about computed instances. Even if this last result is weaker than other related results [16], we point out that it was obtained by means of of a general purpose verification method based on declarative semantics, whilst in [16] the results are achieved through operational reasoning.
We believe that the obtained results illustrate the broad applicability of the adopted verification principles.

Acknowledgements

This work has been partially supported by Esprit BRA 6810 *Compulog2*. We wish to thank the anonymous referees for many helpful comments.

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 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.

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

[4] K.R. Apt. Program Verification and Prolog. In E. Börger, editor, *Specification and Validation methods for Programming languages and systems.* Oxford University Press, 1994.

[5] K.R. Apt, M. Gabbrielli, and D. Pedreschi. A Closer Look at Declarative Interpretations. Technical Report CS-R9470, Centre for Mathematics and Computer Science, Amsterdam, 1994. To appear in *Journal of Logic Programming.*

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

[7] K.R. Apt and D. Pedreschi. Studies in Pure Prolog: Termination. In J. W. Lloyd, editor, *Symposium on Computational Logic*, pages 150–176. Springer-Verlag, Berlin, 1990.

[8] K.R. Apt and F. Turini, editors. *Meta-logics and Logic Programming.* The MIT Press, 1995.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Hierarchies through Basic Metalevel Operators. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 381–396, 1990.

[14] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for Modularising Logic Programming. In A. Pettorossi, editor, *Meta-programming in Logic*, number 649 in LNCS, pages 105–119. Springer-Verlag, 1992.

[15] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, 1994.

[16] A. Brogi and F. Turini. Semantics of meta-logic in an algebra of programs. In S.Abramsky, editor, *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, 1994.

[17] A. Brogi and F. Turini. Meta-logic for program composition: semantic issues. In K.R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 83–110. The MIT Press, 1995.

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

[19] S. Costantini and G. A. Lanzarone. A metalogic programming approach: Language, semantics and applications. *Journal of Experimental and Theoretical Computer Science*, 6:239–287, 1994.

[20] P. M. Hill and J. W. Lloyd. Analysis of Metaprograms. In H. D. Abramson and M. H. Rogers, editors, *Metaprogramming in Logic Programming*, pages 23–52. The MIT Press, 1989.

[21] Y. Jiang. Ambivalent logic as the semantic basis fo metalogic programming: I. In P. Van Hentenryck, editor, *Proc. of the International Conference on Logic Programming*, pages 387–401. MIT Press, 1994.

[22] 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.

[23] M. Kalsbeek. Personal communication. 1995.

[24] M. Kalsbeek and Y. Jiang. A vademecum of Ambivalent logic. In K.R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 27–56. The MIT Press, 1995.

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

[26] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *Journal of Logic Programming*, 22(1):47–99, 1995.

[27] D. Pedreschi and S. Ruggieri. Termination is language-independent. In M. Alpuente et. al., editor, *Proceedings of the GULP-PRODE'94 Joint Conference on Declarative Programming.* Univ. Politecnica de Valencia, Valencia, 1994.

[28] D. Pedreschi and S. Ruggieri. Verification of Logic Programs. Technical report, Dipartimento di Informatica, Università di Pisa, 1996. To appear.

[29] 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.

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

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

A Proofs

Theorem 26

Proof Let A be in *Pre*.
If $A \in M^L_{V \cup P}$ then, by *(1)* and Theorem 6, $A \in Post \cup Post'$. *(2)* implies that $A \in Post$. By [GC] *(4)*, this implies `demo`$(A) \in Post'$. Noting that, by [GC] *(3)*, `demo`$(A) \in Pre'$ and by the fact that $Post'$ is well-supported we have `demo`$(A) \in M^L_V$. Finally, as $M^L_V \subseteq M^L_{V \cup P}$, we conclude that `demo`$(A) \in M^L_{V \cup P}$.

Conversely, let `demo`(A) be in $M^L_{V \cup P}$. By [GC] *(1,3)* and Theorem 6, it results that `demo`$(A) \in Post \cup Post'$, and by *(3)*, `demo`$(A) \in Post'$. By [GC] *(3,4)*, this implies $A \in Post$. Since *Post* is well-supported, we conclude that $A \in M^L_P \subseteq M^L_{V \cup P}$.

This shows that $V \cup P$ is a safe amalgamation for V and P, *Pre*. □

Theorem 28

Proof With the hypothesis of the Theorem, one can easily show that

$$\vdash \{Pre \cup Pre_{VU(P)}\}\ P \cup VU(P)\ \{Post \cup Post_{VU(P)}\}. \tag{13}$$

with $Post \cup Post_{VU(P)}$ well-supported w.r.t. $P \cup VU(P)$ and $Pre \cup Pre_{VU(P)}$. Supposing to prove that $VU(P)$ satisfies [GC] with respect to $VU(P) \cup P$ and $Pre \cup Pre_{VU(P)}$, Theorem 20 allows to conclude that $VU(P)$ is *declaratively correct* with respect to $VU(P) \cup P$ and $Pre \cup Pre_{VU(P)}$. As a consequence, it is *declaratively correct* with respect to $VU(P) \cup P$ and *Pre* as well.

Therefore, to conclude the proof, we only need to show that $VU(P)$ satisfies [GC].
Let us define Pre' and $Post'$ as $Pre_{VU(P)}$ and $Post_{VU(P)}$ respectively.
It is readily checked, under the hypothesis of the theorem and (13), that

$$\vdash \{Pre'\}\ VU(P)\ \{Post'\}$$

and that $Post'$ is well-supported w.r.t. $VU(P)$ and Pre', i.e. [GC] *(1,2)*.
In addition, for $A \in Pre \cup Pre_{VU(P)}$ we have directly from the definitions of Pre', $Post'$ and from Definition 27, that:
[GC] *(3)* `demo`$(A) \in Pre'$, and
[GC] *(4)* $A \in Post \cup Post_{VU(P)}$ iff `demo`$(A) \in Post'$. □

Lemma 45

Proof We recall the Theorem on Constants (see [30]), stating that

$$T \models_L \phi \quad \text{iff} \quad T \models_{L'} \phi^c \tag{14}$$

where T is a first order theory T, and ϕ^c is obtained from ϕ by replacing (free) variables with constants in L' but not in L.

if) Consider A such that $Pre \models A$. By (14), we have that $P \models_L A$ iff $P \models_{L'} A^c$. Since A^c is ground, this is equivalent to $A^c \in M^{L'}_P$. By declarative correctness, and noting that

$A^c \in Pre^u$, we have $A^c \in M_P^{L'}$ iff $\mathtt{demo}(A^c) \in M_V^{L'}$. Since A^c is ground, this is equivalent to $V \models_{L'} \mathtt{demo}(A^c)$. Finally, by (14), $V \models_{L'} \mathtt{demo}(A^c)$ is equivalent to $V \models_L \mathtt{demo}(A)$.
Therefore, we showed $P \models_L A$ iff $V \models_L \mathtt{demo}(A)$ for every A such that $Pre \models A$, i.e. V is $\mathcal{C}$-correct w.r.t. P and Pre.

only if) First of all, note that every atom in Pre^u can be written in the form A^c for some $A \in Atom_L$. Consider now $A^c \in Pre^u$. Since A^c is ground, it results that $A^c \in M_P^{L'}$ iff $P \models_{L'} A^c$. By (14), $P \models_{L'} A^c$ iff $P \models_L A$, and by the hypothesis of $\mathcal{C}$-correctness $P \models_L A$ iff $V \models_L \mathtt{demo}(A)$. By (14) again, this is equivalent to $V \models_{L'} \mathtt{demo}(A^c)$. Since A^c is ground, $V \models_{L'} \mathtt{demo}(A^c)$ iff $\mathtt{demo}(A^c) \in M_V^{L'}$.
In conclusion, we showed that $A \in M_P^{L'}$ iff $\mathtt{demo}(A) \in M_V^{L'}$ for every $A \in Pre^u$, i.e. that V is declaratively correct (on L') w.r.t. P and Pre^u. □