Loop Checking and the Well-Founded Semantics

Vladimir Lifschitz[1] **and Norman McCain**[1]
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA
{vl,mccain}@cs.utexas.edu

Teodor C. Przymusinski[2]
Department of Computer Science
University of California at Riverside
Riverside, CA 92521, USA
teodor@cs.ucr.edu

Robert F. Stärk
Department of Mathematics
Stanford University
Stanford, CA 94305, USA
staerk@gauss.stanford.edu

Abstract. Using a calculus of goals, we define the success and failure of a goal for propositional programs in the presence of loop checking. The calculus is sound with respect to the well-founded semantics; for finite programs, it is also complete. A Prolog-style proof search strategy for a modification of this calculus provides a query evaluation algorithm for finite propositional programs under the well-founded semantics. This algorithm is implemented as a meta-interpreter.

1 Introduction

A "loop check" in a logic programming system is a mechanism that allows the system to avoid some infinite loops. For instance, the execution of the program

$$\begin{array}{l} p \leftarrow not\ q, not\ r, \\ q \leftarrow r, p, \\ r \leftarrow q, p \end{array} \tag{1}$$

under Prolog would lead to an infinite loop, because Prolog would attempt to "derive" q from r and then r from q. A system with a loop check may be able to recognize this fact and to conclude that the goals q and r fail, and, accordingly, the goal p succeeds.

In this paper, we make the idea of loop checking precise and define mathematically what is meant by the success and failure of a goal in the presence of

[1] Partially supported by the National Science Foundation grant #IRI-9306751.
[2] Partially supported by the National Science Foundation grant #IRI-9313061.

loop checking. On the basis of this definition, we prove the soundness and completeness of query evaluation with loop checking for finite propositional programs under the well-founded semantics [Van Gelder *et al.*, 1990], [Przymusinski, 1991]. It follows that loop checking is also sound for the answer set ("stable model") semantics [Gelfond and Lifschitz, 1991].

The mathematical theory of loop checking in logic programming is the subject of a recent dissertation [Bol, 1991]. In this note, we limit attention to a particularly simple case, propositional programs; the problems investigated by Bol have to do, for the most part, with nonground queries and thus do not even arise in the propositional case. On the other hand, Bol's results are restricted to (locally) stratified programs, and in this respect our theory is more general. For example, it applies to program (1), which is not stratified.

Our definition of loop checking is in terms of a "calculus of goals," and in this sense we extend here the earlier work that relates Prolog to the "SLDNF calculus" [Lifschitz, 1995]. The view of Prolog developed in that paper provides a clean separation between the inference rules and the search strategy, which makes the mathematics of Prolog considerably simpler.[3] For instance, the soundness of the calculus from [Lifschitz, 1995] with respect to the completion semantics of [Clark, 1978] can be verified by straightforward induction, and then the soundness of Prolog with respect to that semantics will immediately follow.

The situation with the soundness of loop checking is similar, with an important difference. The existing approaches to the semantics of logic programming can be loosely divided into two categories, depending on how they treat cycles in the program. Consider the simplest cycle: $p \leftarrow p$. Adding a rule like this can affect the meaning of a program under the completion semantics [Clark, 1978] and some of its variants, whereas the well-founded semantics, the answer set semantics and other more recent semantical theories treat rules of this kind as irrelevant. Loop checking is a better match for the second group of theories than for the first. Accordingly, we establish the soundness of loop checking relative to the well-founded semantics and the answer set semantics — representatives of the second group.

In Section 2 we review the SLDNF calculus as defined in [Lifschitz, 1995], and then introduce its modification that is more convenient for our present purposes. The calculus for loop checking and its properties are discussed in Section 3. The calculus turns out to be sound and complete for finite programs under the well-founded semantics. In Section 4, we show how this calculus leads to a query evaluation method for the well-founded semantics, and how this method can be implemented as a meta-interpreter. Section 5 contains proof outlines.

[3] The SLDNF calculus is a reformulation of the inductive characterization of the success and failure of goals under SLDNF given in [Kunen, 1989], and is similar to the calculus proposed earlier in [Mints, 1986]. The idea of separating the search space from the search strategy underlies also the analysis of tabulation techniques in [Bol and Degerstedt, 1993].

2 The SLDNF Calculus

A *rule element* is a (propositional) atom possibly preceded by the negation as failure symbol *not*. [4] A *rule* is a pair $Head \leftarrow Body$, where $Head$ is an atom and $Body$ is a finite set of rule elements. A *program* is a set of rules (not necessarily finite).

2.1 First Formulation

A *goal* is a finite set of rule elements. In the *SLDNF calculus of the first kind* corresponding to a program Π [Lifschitz, 1995], the derivable objects are expressions of the forms $\models G$ ("G succeeds") and $=\!\!| \, G$ ("G fails"), where G is a goal. The only axiom is

$$\models \emptyset.$$

In the inference rules, the following abbreviation is used: For any atom A, $Bodies(A)$ is the set of the bodies of all rules in Π with the head A. There are four inference rules:

$$(SP^1)\ \frac{\models G \cup B}{\models G \cup \{A\}} \qquad \text{where } B \in Bodies(A)$$

$$(FP^1)\ \frac{=\!\!|\, G \cup B \qquad \text{for all } B \in Bodies(A)}{=\!\!|\, G \cup \{A\}}$$

$$(SN^1)\ \frac{\models G \qquad =\!\!|\, \{A\}}{\models G \cup \{not\ A\}}$$

$$(FN^1)\ \frac{\models \{A\}}{=\!\!|\, G \cup \{not\ A\}}$$

In the names of the inference rules, S stands for *success*, F for *failure*, P for *positive*, and N for *negative*. Note that the number of premises of (FP^1) equals the cardinality of $Bodies(A)$. In particular, it can be zero or infinite.

As an example, consider the program

$$\begin{aligned} &p, \\ &q \leftarrow p, not\ r, \\ &q \leftarrow r, not\ p, \\ &r \leftarrow p, not\ s. \end{aligned} \tag{2}$$

Here is a derivation of $=\!\!|\, \{q\}$ in the corresponding SLDNF calculus of the first kind. To simplify notation, we drop the braces in expressions of the forms $\models \{\ldots\}$ and $=\!\!|\, \{\ldots\}$.

---

[4] We do not use here the term "literal," in order to avoid conflict with the terminology of [Gelfond and Lifschitz, 1991] and [Kowalski and Sadri, 1991], where literals are formed using classical (explicit) negation.

$$(FP^1)\,\dfrac{(FN^1)\,\dfrac{(SP^1)\,\dfrac{(SP^1)\,\dfrac{(SN^1)\,\dfrac{\models \emptyset \qquad (FP^1)\,\dfrac{}{\dashv s}}{\models \mathit{not}\ s}}{\models p, \mathit{not}\ s}}{\models r}}{\dashv p, \mathit{not}\ r} \qquad (FN^1)\,\dfrac{(SP^1)\,\dfrac{\models \emptyset}{\models\ p}}{\dashv r, \mathit{not}\ p}}{\dashv q}$$

Since the SLDNF calculus is sound relative to each of the three semantics of negation as failure mentioned in the introduction, this derivation shows that, under any of them, program (2) makes q false.

As observed in [Lifschitz, 1995], Prolog can be viewed as proof search in the SLDNF calculus of the first kind.

2.2 Second Formulation

In the *SLDNF calculus of the second kind* corresponding to a program Π, the derivable objects are expressions of the forms $\models E$ and $\dashv E$, where E is a rule element. There are no axioms.

In order to describe the inference rules, we need the following definition. For any atom A, a *cover* of A is a set of rule elements that has a nonempty intersection with each set in $Bodies(A)$. It is clear that a superset of a cover is a cover; we will be interested in the covers of A that are minimal relative to set inclusion. The set of such covers will be denoted by $Covers(A)$. For instance, in the case of program (2),

$$\begin{array}{l} Covers(q) = \{\{p, r\}, \{p, \mathit{not}\ p\}, \{\mathit{not}\ r, r\}, \{\mathit{not}\ r, \mathit{not}\ p\}\}, \\ Covers(p) = \emptyset. \end{array}$$

The calculus has four inference rules:

$$(SP^2)\ \dfrac{\models E \qquad \text{for all } E \in B}{\models A} \qquad \text{where } B \in Bodies(A)$$

$$(FP^2)\ \dfrac{\dashv E \qquad \text{for all } E \in C}{\dashv A} \qquad \text{where } C \in Covers(A)$$

$$(SN^2)\ \dfrac{\dashv A}{\models \mathit{not}\ A}$$

$$(FN^2)\ \dfrac{\models A}{\dashv \mathit{not}\ A}$$

Note that any application of (SP^2) has finitely many premises, possibly zero. The number of premises of (FP^2) can be infinite. It is clear that, for any atom A, $\models \mathit{not}\ A$ is derivable in this calculus iff $\dashv A$ is derivable.

Here is a derivation of $\dashv q$ in the SLDNF calculus of the second kind for program (2):

$$
(FP^2)\,\dfrac{(FN^2)\,\dfrac{(SP^2)\,\dfrac{(SP^2)\,\dfrac{}{\models p} \qquad (SN^2)\,\dfrac{(FP^2)\,\dfrac{}{\dashv s}}{\models \mathit{not}\ s}}{\models r}}{\dashv \mathit{not}\ r} \qquad (FN^2)\,\dfrac{(SP^2)\,\dfrac{}{\models p}}{\dashv \mathit{not}\ p}}{\dashv q}
$$

We will denote the two forms of the SLDNF calculus for a program Π by SLDNF^1_Π and SLDNF^2_Π. The two formulations are equivalent:

Theorem 1. *For any program Π and any goal G,*

- *expression $\models G$ is derivable in SLDNF^1_Π iff for every $E \in G$, $\models E$ is derivable in SLDNF^2_Π,*
- *expression $\dashv G$ is derivable in SLDNF^1_Π iff for some $E \in G$, $\dashv E$ is derivable in SLDNF^2_Π.*

The second formulation is more economical, in the sense that its derivable expressions have a simpler structure. On the other hand, it is not clear how to extend it to programs with variables.

The calculus of goals for loop checking described in Section 3.1 below is analogous to the second form of the SLDNF calculus.

2.3 Prolog

As we mentioned in Section 2.1, Prolog can be viewed as a proof search strategy for the SLDNF calculus of the first kind. Alternatively, we can think of Prolog as a proof search strategy for the second formulation, as follows.

Consider a finite program Π and a rule element E. We would like to "evaluate" E, that is, to find a derivation of one of the expressions $\models E$, $\dashv E$ in the calculus SLDNF^2_Π. The process is recursive, and it terminates only if $\models E$ or $\dashv E$ is derivable.

Case 1: E is an atom A. Consider, one by one, the sets $B \in \mathit{Bodies}(A)$, and evaluate the elements of each, until a set B is found such that $\models F$ is derived for every element F of B; whenever a set B is determined not to have this property, note for which element F_B of B the expression $\dashv F_B$ is derived. If such a B is found, then $\models A$ can be derived by (SP^2). If all sets $B \in \mathit{Bodies}(A)$ have been checked and none turned out to have this property, then an element F_B is selected in each $B \in Bodies(A)$. Then the set $\{F_B \mid B \in Bodies(A)\}$ is a cover of A, and $\Dashv A$ can be derived by (FP^2).

Case 2: E has the form *not* A. Then evaluate A. If $\models A$ is derived, then $\Dashv E$ can be derived by (FN^2). If $\Dashv A$ is derived, then $\models E$ can be derived by (SN^2).

Note that the objects that we operate with in the process of search are rule elements E, rather than derivable expressions $\models E$, $\Dashv E$. Which of these two expressions will be eventually derived is not determined until the search is completed. This is what we call "Prolog-style" search. This organization of proof search is possible because of a close similarity between the inference rules for success (SP^2), (SN^2) on the one hand, and for failure (FP^2), (FN^2) on the other. The same similarity between success and failure rules will be found in the calculi for loop checking described in Sections 3.1 and 4.1, and for this reason we will be able to extend to those systems the idea of Prolog-style search.

3 Loop Checking

3.1 Calculus for Loop Checking

We will now define the calculus for loop checking corresponding to a given program Π. The derivable expressions of this calculus have the forms $\models E$ and $\Dashv E\ [X]$, where E is a rule element and X a finite set of atoms. We will drop the annotation $[X]$ in $\Dashv E\ [X]$ if X is empty. Intuitively, the expression $E\ [X]$ represents the task of "establishing" E by the rules of the program without having established any of the elements of X in the process — without "passing through" X. Consider, for instance, the program

$$\begin{array}{l} p, \\ q \leftarrow p. \end{array}$$

One can perform the task represented by q, but not the task represented by $q\ [p]$.

The axioms of the calculus are the expressions

$$\Dashv A\ [X] \qquad (A \in X).$$

(If $A \in X$, then it is impossible to establish A without having established at least one element of X.) The inference rules are:

$$(SP^L)\ \frac{\models E \qquad \text{for all } E \in B}{\models A} \qquad \text{where } B \in Bodies(A)$$

$$(FP^L)\ \frac{\Dashv E\ [X \cup \{A\}] \qquad \text{for all } E \in C}{\Dashv A\ [X]} \qquad \text{where } C \in Covers(A)$$

$$(SN^L)\ \frac{\Dashv A}{\models \mathit{not}\ A}$$

$$(FN^L)\ \frac{\models A}{\Dashv \mathit{not}\ A\ [X]}$$

We will denote this calculus by L_Π.

Note that the success rules (SP^L) and (SN^L) are the same as (SP^2) and (SN^2). Rule FP^L says that it is impossible to establish A without passing through X if it is impossible to establish all elements of a cover of A without passing through $X \cup \{A\}$. Here, by adding A to X, we eliminate loops in the process of establishing A.

In the following example, Π is program (1). Here is a derivation of $\models p$:

$$
(SP^L)\ \dfrac{
(SN^L)\ \dfrac{(FP^L)\ \dfrac{(FP^L)\ \dfrac{\dashv q\ [q,r]}{\dashv r\ [q]}}{\dashv q}}{\models \mathit{not}\ q}
\qquad
(SN^L)\ \dfrac{(FP^L)\ \dfrac{(FP^L)\ \dfrac{\dashv r\ [q,r]}{\dashv q\ [r]}}{\dashv r}}{\models \mathit{not}\ r}
}{\models p}
$$

To simplify notation, we drop the braces in expressions of the form $[\{\ldots\}]$.

3.2 Soundness and Completeness Theorems

The calculus L_Π is sound for the well-founded semantics:

Theorem 2. *For any program Π and any atom A,*

- *if $\models A$ is derivable in L_Π then A belongs to the well-founded model of Π,*
- *if $\dashv A$ is derivable in L_Π then $\neg A$ belongs to the well-founded model of Π.*

It follows that L_Π is "consistent":

Corollary 1. *For any program Π, there is no rule element E such that both $\models E$ and $\dashv E$ are derivable in L_Π.*

It is possible, of course, that, for some atom A, neither $\models A$ nor $\dashv A$ is derivable in L_Π. Theorem 2 shows that this happens whenever the truth value of A in the well-founded semantics of Π is *undefined.* The simplest example is

$$p \leftarrow \mathit{not}\ p. \tag{3}$$

In the calculus corresponding to this program, one can derive neither $\models p$ nor $\dashv p$.

Since the answer set semantics is stronger than the well-founded semantics [Przymusinski, 1990], Theorem 2 implies that L_Π is sound for the answer set semantics also:

Corollary 2. *For any program Π and any atom A,*

- *if $\models A$ is derivable in L_Π then A belongs to all answer sets for Π,*
- *if $\dashv A$ is derivable in L_Π then A does not belong to any of the answer sets for Π.*

If Π is finite then L_Π is complete for the well-founded semantics:

Theorem 3. *For any finite program Π and any atom A,*

- *expression $\models A$ is derivable in L_Π iff A belongs to the well-founded model of Π,*
- *expression $\dashv A$ is derivable in L_Π iff $\neg A$ belongs to the well-founded model of Π.*

For infinite programs, the assertion of Theorem 3 is generally invalid. Consider, for instance, the program

$$p_n \leftarrow p_{n+1} \qquad (n \geq 0).$$

The expression $\dashv p_0$ is not derivable in the corresponding calculus for loop checking, although $\neg p_0$ belongs to the well-founded model of the program.

3.3 Query Evaluation in the Calculus for Loop Checking

The Prolog proof search strategy (Section 2.3) can be extended to the calculus L_Π as follows.

Consider a finite program Π, a rule element E, and a finite set X of atoms. To "evaluate" the expression $E\ [X]$ means now to find a derivation of one of the expressions $\models E$, $\dashv E\ [X]$ in the calculus L_Π.

Case 1: E is an atom A. If $A \in X$ then $\dashv A\ [X]$ is an axiom. Otherwise, consider, one by one, the sets $B \in \mathit{Bodies}(A)$, and, for the successive elements F of each of these sets, evaluate $F\ [X \cup \{A\}]$, until a set B is found such that $\models F$ is derived for every element F of B; whenever a set B is determined not to have this property, note for which element F_B of B the expression $\dashv F_B\ [X \cup \{A\}]$ has been derived. If such a B is found, then $\models A$ can be derived by (SP^L). If all sets $B \in \mathit{Bodies}(A)$ have been checked and none turned out to have this property, then an element F_B is selected in each $B \in \mathit{Bodies}(A)$. Then the set $\{F_B \mid B \in \mathit{Bodies}(A)\}$ is a cover of A, and $\dashv A\ [X]$ can be derived by (FP^L).

Case 2: E has the form *not* A. Then evaluate A. If $\models A$ is derived, then $\dashv E\ [X]$ can be derived by (FN^L). If $\dashv A$ is derived, then $\models E$ can be derived by (SN^L).

For example, when this process is applied to program (1) as Π, the atom p as E, and the empty set as X, the result is the derivation of $\models p$ shown at the end of Section 3.1.

As another example, consider the program

$$\begin{array}{l} p \leftarrow p, \\ p. \end{array}$$

We want to evaluate p. Using the first rule, we are led to evaluate the expression $p\ [p]$, and observe that $\dashv p\ [p]$ is an axiom. Hence, we go on to try the second rule of the program. Since the body of the rule is empty, $\models p$ is derivable by (SP^L) from the empty set of premises. The following derivation is found:

$$(SP^L)\ \frac{}{\models p}$$

It is interesting to observe that this figure is also a derivation in the SLDNF calculus of the second kind, but Prolog will not find this derivation. The reason is that Prolog will loop on the first rule of the program, and the second rule will never be tried.

Like Prolog, this query evaluation procedure does not necessarily terminate. For instance, it does not terminate whenever applied to an atom A (and the empty X) such that neither $\models A$ nor $\dashv A$ is derivable in L_Π; program (3) is an example. Moreover, it is possible that $\models A$ is derivable but the procedure does not terminate anyway. This is illustrated by the program

$$\begin{array}{l} p \leftarrow not\ q, \\ q \leftarrow not\ p, \\ q \leftarrow not\ r. \end{array} \tag{4}$$

The expression $\models q$ is derivable in the corresponding calculus for loop checking:

$$\begin{array}{ll} (FP^L) & \overline{} \\ & \quad \dashv r \\ (SN^L) & \overline{} \\ & \models not\ r \\ (SP^L) & \overline{} \\ & \quad \models q \end{array}$$

However, given the atom q as input, the evaluation strategy described above will not find this derivation. The reason is that the algorithm loops infinitely through the following sequence of expressions:

$$q,\ not\ p,\ p,\ not\ q,\ q,\ not\ p, \ldots\ .$$

4 A Prolog-Style Evaluation Algorithm for the Well-Founded Semantics

We have seen that, for any finite program Π, the calculus L_Π is sound and complete with respect to the well-founded semantics of Π, but Prolog-style query evaluation in this calculus does not terminate in some cases. For this reason, Prolog-style proof search in L_Π does not provide a query evaluation algorithm for finite propositional programs under the well-founded semantics.

In this section we describe a simple modification of L_Π that has essentially no effect on its deductive capabilities but causes the evaluation method to always terminate. The algorithm based on the modified calculus is implemented below as a meta-interpreter.

4.1 Modified Calculus

The new calculus corresponding to a given program Π will be denoted by $\widetilde{L}_\Pi$. Its derivable expressions have the forms $\models E\ [Y]$ and $\dashv E\ [X;Y]$, where E is a rule element and X, Y are finite sets of atoms. We will drop the braces around the elements of X and Y, and will drop the annotations $[\emptyset]$ and $[\emptyset; \emptyset]$ altogether.

The axioms of $\widetilde{L}_\Pi$ are the expressions

$$=\!\!| A\ [X;Y] \qquad (A \in X).$$

The inference rules are:

$$(SP^{\widetilde{L}})\ \frac{\models E\ [Y] \qquad \text{for all } E \in B}{\models A\ [Y]} \qquad \text{where } B \in \mathit{Bodies}(A)$$

$$(FP^{\widetilde{L}})\ \frac{=\!\!| E\ [X \cup \{A\}; Y] \qquad \text{for all } E \in C}{=\!\!| A\ [X;Y]} \qquad \text{where } C \in \mathit{Covers}(A)$$

$$(SN^{\widetilde{L}})\ \frac{=\!\!| A\ [\emptyset; Y \cup \{A\}]}{\models \mathit{not}\ A\ [Y]} \qquad \text{where } A \notin Y$$

$$(FN^{\widetilde{L}})\ \frac{\models A\ [Y \cup \{A\}]}{=\!\!| \mathit{not}\ A\ [X;Y]} \qquad \text{where } A \notin Y$$

The restriction $A \notin Y$ in the last two rules guarantees that, in any branch of a derivation, these rules will not be applied to the same A more than once.

Here is a derivation in the modified calculus for program (1):

$$\dfrac{(SN^{\widetilde{L}})\ \dfrac{(FP^{\widetilde{L}})\ \dfrac{(FP^{\widetilde{L}})\ \dfrac{=\!\!| q\ [q,r;q]}{=\!\!| r\ [q;q]}}{=\!\!| q\ [;q]}}{\models \mathit{not}\ q} \qquad (SN^{\widetilde{L}})\ \dfrac{(FP^{\widetilde{L}})\ \dfrac{(FP^{\widetilde{L}})\ \dfrac{=\!\!| r\ [q,r;r]}{=\!\!| q\ [r;r]}}{=\!\!| r\ [;r]}}{\models \mathit{not}\ r}}{\models p}\ (SP^{\widetilde{L}})$$

To simplify notation, we drop the braces in expressions of the form $[\{\ldots\};\{\ldots\}]$.

Theorem 4. *For any program Π and any rule element E, each of the expressions $\models E$, $=\!\!| E$ is derivable in $\widetilde{L}_\Pi$ iff it is derivable in L_Π.*

In this sense, the calculus $\widetilde{L}_\Pi$ is equivalent to L_Π. Consequently, L_Π can be replaced by $\widetilde{L}_\Pi$ in the statements of all theorems and corollaries in Section 3.2.

4.2 Query Evaluation in the Modified Calculus

Consider a finite program Π, a rule element E, and finite sets X, Y of atoms. To "evaluate" the expression $E\ [X;Y]$ means now to find a derivation of one of the expressions $\models E\ [Y]$, $=\!\!| E\ [X;Y]$ in the calculus $\widetilde{L}_\Pi$ or to determine that neither expression is derivable. Thus the evaluation of an expression can have one of three possible outcomes (rather than two, as in the algorithms presented in Sections 2.3 and 3.3).

Case 1: E is an atom A. *Case 1.1:* $A \in X$. Then $\dashv A\ [X;Y]$ is an axiom. *Case 1.2:* $A \notin X$. Then consider, one by one, the sets $B \in \mathit{Bodies}(A)$, and, for the successive elements F of each of these sets, evaluate $F\ [X \cup \{A\};Y]$, until a set B is found such that $\models F\ [Y]$ is derived for every element F of B; whenever a set B is determined not to have this property, continue the process of evaluation of the expressions $F\ [X \cup \{A\};Y]$ for the elements F of B until an element F_B of B is found for which $\dashv F_B\ [X \cup \{A\};Y]$ is derived. *Case 1.2.1:* Such a B is found. Then $\models A\ [Y]$ can be derived by $(SP^{\widetilde{L}})$. *Case 1.2.2:* There is no such B, and F_B is selected in every $B \in \mathit{Bodies}(A)$. Then the set $\{F_B \mid B \in \mathit{Bodies}(A)\}$ is a cover of A, and $\dashv A\ [X;Y]$ can be derived by (FP^L). *Case 1.2.3:* There is no such B, and for at least one B, F_B could not be selected. Then neither $\models E\ [Y]$ nor $\dashv E\ [X;Y]$ is derivable.

Case 2: E has the form *not* A. *Case 2.1:* $A \in Y$. Then neither $\models E\ [Y]$ nor $\dashv E\ [X;Y]$ is derivable. *Case 2.2:* $A \notin Y$. Then evaluate $A\ [\emptyset; Y \cup \{A\}]$. *Case 2.2.1:* $\models A\ [Y \cup \{A\}]$ is derived. Then $\dashv E\ [X;Y]$ can be derived by $(FN^{\widetilde{L}})$. *Case 2.2.2:* $\dashv A\ [\emptyset; Y \cup \{A\}]$ is derived. Then $\models E\ [Y]$ can be derived by $(SN^{\widetilde{L}})$. *Case 2.2.3:* Neither $\models A\ [Y \cup \{A\}]$ nor $\dashv A\ [\emptyset; Y \cup \{A\}]$ is derivable. Then neither $\models E\ [Y]$ nor $\dashv E\ [X;Y]$ is derivable.

In Case 1.2, we could have included an obvious optimization: Once it is determined that an element F_B cannot be selected in some $B \in \mathit{Bodies}(A)$, there is no need to continue looking for such elements in the other sets from $\mathit{Bodies}(A)$. This optimization is not implemented in the meta-interpreter presented in Section 4.3.

This algorithm always terminates. Indeed, define the *rank* of an expression $E\ [X;Y]$ to be the ordinal $\omega(n - |Y|) + (n - |X|)$, where n is the number of atoms in Π. In every recursive call of the algorithm, the rank of the expression to be evaluated goes down: In Case 1.2, $|Y|$ stays the same and $|X|$ increases; in Case 2.2, $|Y|$ increases. (In the definition of rank, ω can be replaced by $n + 1$.)

From Theorems 3 and 4 we see that the three possible answers produced by this algorithm applied to an atom A with empty X and Y exactly correspond to the three possible truth values of A in the well-founded semantics.

Consider, for instance, the application of this algorithm to program (3) with the expression p as E and with empty X and Y. (The rank of $E\ [X;Y]$ is $\omega + 1$.) First, we are in Case 1.2 and are led to evaluate *not* $p\ [p;]$ (rank ω). This brings us to Case 2.2 and to the task of evaluating $p\ [;p]$ (rank 1). Now we are in Case 1.2 and turn to the evaluation of *not* $p\ [p;p]$ (rank 0). This is Case 2.1; neither $\models$ *not* $p\ [p]$ nor $\dashv$ *not* $p\ [p;p]$ is derivable. Consequently, neither $\models p\ [p]$ nor $\dashv p\ [;p]$ is derivable (Case 1.2.3). Consequently, neither $\models$ *not* p nor $\dashv$ *not* $p\ [p;]$ is derivable (Case 2.2.3). Consequently, neither $\models p$ nor $\dashv p$ is derivable (Case 1.2.3). This result tells us that the truth value of p in the well-founded semantics is *undefined*.

4.3 Implementation

An implementation of the evaluation strategy described in the previous section is given below. It is implemented in Quintus Prolog as the meta-interpreter solve/4. The procedure wf/2 returns the value of an atom in the well-founded semantics.

The programs executed by the meta-interpreter are composed of ground Prolog clauses with purely conjunctive bodies. The symbol not is used for negation as failure.

```
:- op(600,fx,'not').
:- use_module(library(basics)).

wf(A,Val) :- solve(A,[],[],Val).

solve(not A,_X,Y,Ans) :-
        member(A,Y), !, Ans = und.                        % 2.1
solve(not A,_X,Y,Ans) :-
        !, solve(A,[],[A|Y],An1),                         % 2.2
        neg(An1,Ans).                                     % 2.2.1-2.2.3
solve(A,X,_Y,Ans) :-
        member(A,X), !, Ans = false.                      % 1.1
solve(A,X,Y,Ans) :-
        findall(B,clause(A,B),Bs),
        solve_bodies(Bs,[A|X],Y,Ans).                     % 1.2

solve_bodies([B|Bs],X,Y,Ans) :-
        solve_elements(B,X,Y,An1),
        ( An1 = true -> Ans = true
        ; solve_bodies(Bs,X,Y,An2),
          or(An1,An2,Ans) ).                              % 1.2.1-1.2.3
solve_bodies([],_X,_Y,false).

solve_elements((F,Fs),X,Y,Ans) :-
        !, solve(F,X,Y,An1),
        ( An1 = false -> Ans = false
        ; solve_elements(Fs,X,Y,An2),
          and(An1,An2,Ans) ).
solve_elements(F,X,Y,Ans) :-
        F = true -> Ans = true ; solve(F,X,Y,Ans).

and(false,_U,false).
and(und,U,V) :-
        U = false -> V = false ; V = und.
and(true,U,U).

or(false,U,U).
```

```
or(und,U,V) :-
        U = true -> V = true ; V = und.
or(true,_U,true).

neg(false,true).
neg(und,und).
neg(true,false).
```

5 Proofs

5.1 Proof of Theorem 1

Lemma 1. *For any program Π and any goals G_1, G_2, if $\models G_1$ and $\models G_2$ are derivable in $SLDNF^1_\Pi$ then so is $\models G_1 \cup G_2$.*

The proof is by induction on the derivation of $\models G_1$.

Lemma 2. *For any program Π and any goals G_1, G_2 such that $G_1 \subset G_2$, if $\dashv G_1$ is derivable in $SLDNF^1_\Pi$ then so is $\dashv G_2$.*

The proof is by induction on the derivation of $\dashv G_1$.

Proof of Theorem 1. Left-to-right, both parts of the theorem can be proved simultaneously by induction on the given derivation in $SLDNF^1_\Pi$. Right-to-left, both parts can be proved simultaneously by induction on the given derivation in $SLDNF^2_\Pi$, using Lemmas 1 and 2.

5.2 A Characterization of the Well-Founded Model

For a set X of atoms, define

$$\begin{aligned} \neg X &= \{\neg A \ : \ A \in X\}, \\ not(X) &= \{not\ A \ : \ A \in X\}. \end{aligned}$$

Every set of rule elements can be represented as $Pos \cup not(Neg)$, where Pos and Neg are sets of atoms.

The proofs of Theorems 2 and 3 are based on the following characterization of the well-founded semantics.

Lemma 3. *The well-founded model of a program Π is the set $I \cup \neg J$, where I and J are the least sets of atoms with the following two properties:*

(WF1) If $A \leftarrow Pos \cup not(Neg)$ is a rule of Π such that $Pos \subset I$ and $Neg \subset J$ then $A \in I$.
(WF2) There exists a mapping ν from atoms to natural numbers such that for each atom $A \notin J$ there exists a rule $A \leftarrow Pos \cup not(Neg)$ in Π for which
  (a) for all $B \in Pos$, $\nu(B) < \nu(A)$ and $B \notin J$,
  (b) $Neg \cap I = \emptyset$.

Proof. The well-founded model of a program can be described as the least fixpoint of an operator Ψ associated with this program ([Przymusinski, 1994], Theorem 13). If I and J satisfy (WF1) and (WF2) then $I \cup \neg J$ is a pre-fixpoint of Ψ. If $I \cup \neg J$ is a fixpoint of Ψ then I and J satisfy (WF1) and (WF2).

5.3 Proofs of Theorems 2 and 3

Proof of Theorem 2. Let I, J and ν be as in (WF1) and (WF2). Then the following can be shown by induction on the derivation:

1. If $\models A$ is derivable in L^{Π} then $A \in I$.
2. If $\models \mathit{not}\ A$ is derivable in L^{Π} then $A \in J$.
3. If $\dashv A\ [X]$ is derivable in L^{Π} and $\nu(A) < \nu(B)$ for all $B \in X$ then $A \in J$.
4. If $\dashv \mathit{not}\ A\ [X]$ is derivable in L^{Π} then $A \in I$.

Now the assertion of the theorem follows by Lemma 3.

Proof of Theorem 3. In this proof, by an atom we mean an atom occurring in Π. Let n be the number of atoms. Let I be the set of atoms A such that $\models A$ is derivable in L_{Π}. Let J be the set of atoms A such that $\dashv A$ is derivable in L_{Π}. By Lemma 3, it is sufficient to check that I and J satisfy (WF1) and (WF2).

(WF1) Assume that $A \leftarrow \mathit{Pos} \cup \mathit{not}(\mathit{Neg})$ is a rule of Π such that $\mathit{Pos} \subset I$ and $\mathit{Neg} \subset J$. From the definitions of I and J we see that $\models A$ can be derived by rule (SP^L). Thus $A \in I$.

(WF2) For every atom $A \notin J$, let X_A be a set of maximal cardinality such that $\dashv A\ [X_A]$ is not derivable. For every atom $A \notin J$, define $\nu(A) = n - |X_A|$; for $A \in J$, set $\nu(A) = 0$. Take any $A \notin J$. Then $\dashv A\ [X_A]$ is not derivable. In particular, this expression is not an axiom, so that $A \notin X_A$. Moreover, it cannot be derived by (FP^L), that is, there exists a rule $A \leftarrow \mathit{Pos} \cup \mathit{not}(\mathit{Neg})$ in Π such that $\dashv E\ [X_A \cup \{A\}]$ is not derivable for each $E \in \mathit{Pos} \cup \mathit{not}(\mathit{Neg})$. (a) Take any $B \in \mathit{Pos}$. Then $\dashv B\ [\emptyset]$ is not derivable and thus $B \notin J$. We need to show that $\nu(B) < \nu(A)$. Since $\dashv B\ [X_A \cup \{A\}]$ is not derivable, it follows by the definition of X_B that $|X_A \cup \{A\}| \leq |X_B|$. Since $A \notin X_A$, we conclude that $|X_A| < |X_B|$ and thus $\nu(B) = n - |X_B| < n - |X_A| = \nu(A)$. (b) Take any $B \in \mathit{Neg}$. Then $\dashv \mathit{not}\ B$ is not derivable, and therefore $\models B$ is not derivable either, so that $B \notin I$. Thus $\mathit{Neg} \cap I = \emptyset$.

5.4 Proof of Theorem 4

Consider a derivation of $\models E$ or $\dashv E$ in the calculus $\widetilde{L}_{\Pi}$. Replace every derived expression of the form $\models F\ [Y]$ in this derivation by $\models F$, and every derived expression of the form $\dashv F\ [X; Y]$ by $\dashv F\ [X]$. The result is a derivation in the calculus L_{Π}.

To prove the second half of the theorem, we need the following definition. A derivation in the calculus L_{Π} is *minimal* if any two different nodes in the same branch of this derivation contain different derived expressions. It is clear that every expression derivable in L_{Π} has a minimal derivation.

Consider a minimal derivation of $\models E$ or $\dashv E$ in the calculus L_Π. For any node α in this derivation, let Y_α be the set of atoms A such that one of the expressions $\models$ *not* A, $\dashv$ *not* A $[X]$ for some X occurs in the derivation under the node α. Replace every derived expression of the form $\models F$ in every node α by $\models F\ [Y_\alpha]$, and every derived expression $\dashv F\ [X]$ in a node α by $\dashv F\ [X; Y_\alpha]$. We claim that the result is a derivation in the calculus $\widetilde{L}_\Pi$.

To prove this, we need to check that, for any node α of the given derivation that contains the conclusion $\models$ *not* A of (SN^L) or the conclusion $\dashv$ *not* A $[X]$ of (FN^L), $A \notin Y_\alpha$. Assume that $A \in Y_\alpha$. Then some node α' under α contains either $\models$ *not* A or $\dashv$ *not* $A[X']$ for some X'. Consider the premise of the rule leading to α and the premise of the rule leading to α'. Each of them is either $\models A$ or $\dashv A$. By the minimality of the derivation, they cannot be equal. Consequently, one of the expressions is $\models A$ and the other is $\dashv A$. But this is impossible, by Corollary 1 of Theorem 2.

6 Further Work

A derivation in the calculus L_Π can justify the assertion that an atom A is true in the well-founded semantics (if the derived expression is $\models A$) or that it is false (if the derived expression is $\dashv A$). In a forthcoming paper, we introduce a calculus with additional types of derived expressions that correspond to the properties of being not false (true or undefined) and not true (false or undefined). This calculus, like L_Π, is sound and complete for finite propositional programs.

We plan to extend this work in several directions. First, it needs to be generalized to programs with variables. Second, we would like to describe tabulation [Chen and Warren, 1993] in terms of a calculus of goals. Third, it is interesting to know how such calculi can be made complete relative to the answer set semantics.

References

[Bol and Degerstedt, 1993] Roland Bol and Lars Degerstedt. The underlying search for magic templates and tabulation. In *Logic Programming: Proceedings of the Tenth Int'l Conf. on Logic Programming*, pages 793–811, 1993.

[Bol, 1991] Roland Bol. *Loop Checking in Logic Programming.* PhD thesis, University of Amsterdam, 1991.

[Chen and Warren, 1993] Weidong Chen and David Warren. Query evaluation under the well founded semantics. In *The Twelfth ACM Symposium on Principles of Database Systems*, 1993.

[Clark, 1978] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Kowalski and Sadri, 1991] Robert Kowalski and Fariba Sadri. Logic programs with exceptions. *New Generation Computing*, 9:387–400, 1991.

[Kunen, 1989] Kenneth Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7(3):231–245, 1989.

[Lifschitz, 1995] Vladimir Lifschitz. SLDNF, constructive negation and grounding. In *Proc. of ICLP-95*, 1995. To appear.

[Mints, 1986] Grigori Mints. A complete calculus for pure Prolog. *Proc. Academy of Sciences of Estonian SSR*, 35(4):367–380, 1986. In Russian.

[Przymusinski, 1990] Teodor Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, pages 445–464, 1990.

[Przymusinski, 1991] Teodor Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991.

[Przymusinski, 1994] Teodor Przymusinski. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:141–187, 1994.

[Van Gelder *et al.*, 1990] Allen Van Gelder, Kenneth Ross, and John Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, pages 221–230, 1990.