SLDNF, Constructive Negation and Grounding

Vladimir Lifschitz
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA
vl@cs.utexas.edu

Abstract

In some approaches to the declarative semantics of logic programming, a program Π is treated as shorthand for the propositional program $Ground(\Pi)$ whose rules are the ground instances of the rules of Π. We relate the success and failure of a program Π with a goal G under SLDNF with constructive negation to the success and failure of $Ground(\Pi)$ with ground instances of G under propositional SLDNF. As a corollary, we get a soundness theorem for SLDNF with constructive negation relative to the answer set semantics. Since this semantics is closely related to default logic and autoepistemic logic, proofs of such theorems help us understand why the computational mechanisms developed in logic programming provide partial implementations of these nonmonotonic formalisms.

1 Introduction

In the approaches to the declarative semantics of logic programming developed over the last years, variables in a logic program are often treated as metavariables for ground terms. From this perspective, a program is essentially a propositional object. Having developed a theory of propositional programs, we can define the syntax of "schematic programs" — programs with variables — and declare that a schematic program has the same meaning as the propositional program obtained from it by "grounding," that is, by replacing every rule with all its ground instances.

The theory of query evaluation, on the other hand, would lose much of its interest if we restricted it to the case of propositional programs. The use of unification, for instance, or the problem of nonground negative queries, would not even make sense in this context. It is not surprising that the idea of grounding has played a limited role in the development of query evaluation procedures. The semantic idea that has had most influence on that work is the "nonpropositional" idea of predicate completion. For instance, the first soundness theorem for the SLDNF query evaluation procedure expressed its soundness with the respect to the completion semantics [6]. An important paper on constructive negation is entitled *Constructive Negation Based on the Completed Database* [5].

In this note, we try to bridge these two directions in the theory of logic programming by developing a view of SLDNF and constructive negation that stresses the relationship between a schematic program Π and the corresponding propositional program $Ground(\Pi)$. The "grounding theorem" established here relates the success and failure of a goal G with respect to Π to the success and failure of the ground instances of G with respect to $Ground(\Pi)$. This theorem shows that, for any declarative semantics based on grounding, the soundness of SLDNF with a certain form of constructive negation is a consequence of the soundness of propositional SLDNF with respect to the same semantics.

As a corollary, we get a soundness theorem for SLDNF with constructive negation relative to the answer set ("stable model") semantics proposed by Gelfond and Lifschitz [11], [12]. The answer set semantics follows the ideas of [3] and [10], and treats a logic program essentially as a default theory in the sense of [18] or as an autoepistemic theory in the sense of [17]. Proofs of soundness theorems like this are interesting because they explain why the computational mechanisms developed in logic programming provide partial implementations of these nonmonotonic formalisms.

Our definition of SLDNF is a reformulation of the remarkably simple definition due to Kunen [13]. Kunen defined the notion of a computed answer substitution and the notion of a failed query directly, without first introducing SLDNF derivation trees — quite complex combinatorial objects (as done, for instance, in Section 15 of [15] and in Section 2 of [2]). The closure conditions in Kunen's definition can be represented as inference rules. In this way, we arrive at the "SLDNF calculus" corresponding to a given logic program.[1] The advantage of this approach is that it distinguishes between SLDNF as a calculus on the one hand, and proof procedures for this calculus, such as Prolog, on the other. Mathematically, the calculus is much simpler than the process of search involved in the proof procedures, and many questions that one would like to ask about SLDNF are about the calculus itself, and not about search.

Sections 2 and 3 of this paper are mostly devoted to a detailed review of Kunen's approach to SLDNF. In addition, a soundness theorem for the propositional SLDNF relative to the answer set semantics is presented in Section 2.2, and a special case of the grounding theorem in Section 3.3. In Section 4, we define an SLDNF calculus with constructive negation and state the general grounding theorem.

2 Propositional Case

2.1 Propositional SLDNF

We begin with a nonempty set of propositional atoms. A *normal program* is a set of *normal rules* — pairs $Head \leftarrow Body$, where $Head$ is an atom and $Body$ is a finite set whose elements are atoms possibly preceded by the negation as failure symbol *not*. A *goal* is a finite set whose elements are atoms possibly preceded by *not*; thus a goal is a set that can be the body of a normal rule.

Consider a normal program Π. In the *SLDNF calculus* for Π, 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:

$$(SP_p)\ \frac{\models G \cup B}{\models G \cup \{A\}} \qquad \text{if } B \in Bodies(A)$$

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

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

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

In the names of the inference rules, S stands for *success*, F for *failure*, P for *positive*, N for *negative*, and the subscript p indicates that this is a rule of the *propositional* version of the calculus. Note that the number of premises of (FP_p) equals the cardinality of $Bodies(A)$. In particular, it can be zero or infinite.[2]

If $\models G$ is derivable in the SLDNF calculus for Π then we say that G *succeeds* relative to Π. If $=\!\!|\, G$ is derivable then we say that G *fails* relative to Π.

The process that leads from a logic program to the corresponding SLDNF calculus is a nonmonotonic process, in the sense that adding a rule to the program makes one of the SLDNF inference rules, (FP_p), weaker. But there is nothing "nonmonotonic," of course, in the calculus itself.

As an example, consider the program

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

The fact that the goal $\{q\}$ fails relative to this program is demonstrated by the following derivation. To simplify notation, we drop the braces in expressions $\models \{\ldots\}$ and $=\!\!|\, \{\ldots\}$.

$$
(FP_p)\,\dfrac{(FN_p)\,\dfrac{(SP_p)\,\dfrac{(SP_p)\,\dfrac{(SN_p)\,\dfrac{\models \emptyset \qquad (FP_p)\,\dfrac{}{\dashv s}}{\models \mathit{not}\ s}}{\models p, \mathit{not}\ s}}{\models r}}{\dashv p, \mathit{not}\ r} \qquad (FN_p)\,\dfrac{(SP_p)\,\dfrac{\models \emptyset}{\models\ p}}{\dashv r, \mathit{not}\ p}}{\dashv q}
$$

Some queries neither succeed nor fail. It is easy to see, for instance, that p neither succeeds nor fails relative to the program $\{p \leftarrow p\}$.

For any goals G and G', if G fails then $G \cup G'$ fails, too. This is easy to check by induction on the derivation of $\dashv G$. Consequently, adding the "thinning" rule

$$
(T)\ \frac{\dashv G}{\dashv G \cup G'}
$$

to the calculus described above would not change the set of derivable expressions.

2.2 Soundness of Propositional SLDNF

The following theorem expresses the soundness of propositional SLDNF relative to the answer set semantics. It shows that by "evaluating" the goal $\{A\}$ — by showing that it succeeds or that it fails — we learn whether or not A belongs to the answer sets for Π.

Soundness Theorem for Propositional SLDNF. *For any normal program Π and any atom A,*

- *if $\{A\}$ succeeds with respect to Π then A belongs to all answer sets for Π,*

- *if $\{A\}$ fails with respect to Π then A does not belong to any answer set for Π.*

The converse is not true. For instance, we saw that $\{p\}$ does not fail with respect to the program $\{p \leftarrow p\}$, although the only answer set for this program is empty.

2.3 Propositional Prolog

The work of Prolog, in the propositional case, can be viewed as an attempt to evaluate a given goal G by constructing a derivation of one of the expressions $\models G$, $=\!\!\!|\, G$ in the SLDNF calculus by backward chaining. In order to find a rule application that leads to one of these expressions, G is represented as the union of two goals one of which — the "selected subgoal" — is a singleton.

If the selected subgoal does not contain negation as failure, so that G is represented as $G' \cup \{A\}$, then the rule applied in the derivation last will be either (SP_p) or (FP_p). Prolog determines which one by attempting to evaluate the goals $G' \cup B$ for all $B \in \mathit{Bodies}(A)$. If at least one of these goals succeeds then $\models G$ can be derived by (SP_p). If they all fail then $=\!\!\!|\, G$ can be derived by (FP_p).

If the selected subgoal does contain negation as failure, so that G is represented as $G' \cup \{\mathit{not}\ A\}$, then Prolog attempts to evaluate $\{A\}$. If this goal succeeds then $=\!\!\!|\, G$ can be derived by (FN_p). If it fails then Prolog evaluates G'. If it succeeds then $\models G$ will follow by (SN_p). If it fails then $=\!\!\!|\, G$ will follow by (T).

At the beginning of this process, the given program is represented as the list of its rules, with the body of every rule represented as the list of its elements. Similarly, the goal G is represented as a list. The order in which the elements of all these lists are initially arranged has a significant effect on the search process. Specifically, it determines how subgoals are selected, and in what order the elements of $\mathit{Bodies}(A)$ are considered when the bottom-up application of (SP_p) or (FP_p) is attempted.

The Prolog search strategy is incomplete. For instance, given the program

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

with the rules listed in this order, Prolog will never find the one-step derivation of $\models p$ in the SLDNF calculus. Thus Prolog, viewed as a method for establishing that a given atom belongs to all answer sets for a given propositional program, is incomplete in two ways. First, $\models A$ may be not derivable in the SLDNF calculus even though A belongs to all answer sets. Second, Prolog may fail to find a derivation of $\models A$ even though such a derivation exists.

3 SLDNF

In this section, the description of SLDNF from [13] is reproduced in full generality — formulated again in terms of inference rules.

3.1 Schematic Programs

Consider a first-order language **L** without equality that has at least one object constant and at least one predicate constant. A *schematic normal rule*, a *schematic normal program* and a *schematic goal* are defined in the same way as normal rules, normal programs and goals (Section 2.1) except that arbitrary atomic formulas of **L** are used instead of propositional atoms. For any schematic normal rule R, $Ground(R)$ is the set of all ground instances of R. For any schematic normal program Π,

$$Ground(\Pi) = \bigcup_{R \in \Pi} Ground(R).$$

It is clear that $Ground(R)$ and $Ground(\Pi)$ are normal programs in the sense of Section 2.1, if atoms are taken to be the ground atoms of **L**. An *answer set* for a schematic normal program Π is an answer set for $Ground(\Pi)$.

For instance, here is a schematic program in the language that includes the object constants a and b, the unary predicate constants p, q and r, and no function constants:

$$\begin{array}{l} p(a), \\ q(b), \\ r(x) \leftarrow p(x), not\ q(x). \end{array}$$

Call this program Π_1; $Ground(\Pi_1)$ consists of the first two rules and of the two ground instances of the last rule.

3.2 SLDNF for Schematic Programs

In the SLDNF calculus for a schematic normal program Π, the derivable objects are expressions of the forms $\models G : \delta$ and $=\!\mid G$, where G is a schematic goal and δ is a substitution[3] whose support is a subset of the variables of G. If $\models G : \delta$ is derivable then we say that G *succeeds relative to* Π *with computed answer substitution (c.a.s.)* δ; if $=\!\mid G$ is derivable then we say that G *fails relative to* Π.

Recall that, in the propositional case, $Bodies(A)$ was defined as the set of the bodies of all rules of the program whose head is A. For a schematic program, the definition of *Bodies* will be more complicated in two ways. First, a schematic rule can be "applied" to A whenever A can be unified with its head, and we need to know a substitution that unifies them. For this reason, a value of the function *Bodies* will be now a pair — the body of a rule along with a substitution that unifies A with the head of the rule. Second, variable renaming may be needed, so that, instead of a rule R of the given program, we will use its "variant" — a rule of the form $R\theta$, where θ is an invertible substitution.[4] For this reason, the argument of *Bodies* will be not A itself, but a function that tells us, for every rule applicable to A, how its variables are renamed before the unification.

These considerations motivate the following definitions. A schematic rule $H \leftarrow B$ is *applicable* to an atomic formula A if there exist substitutions σ_1 and σ_2 such that $A\sigma_1 = H\sigma_2$. A *matching function* for an atomic formula A and a set V of variables (relative to a schematic program Π) is a function that is defined on the rules $H \leftarrow B$ of Π that are applicable to A and maps every such rule to a pair of substitutions $\langle \theta, \sigma \rangle$ such that

- θ is invertible,
- the variables of $(H \leftarrow B)\theta$ do not belong to V,
- σ is a most general unifier (mgu) for A and $H\theta$.

(If V is finite then such a function always exists.) For any matching function f, $\mathit{Bodies}(f)$ stands for the set of pairs $\langle B\theta, \sigma \rangle$ such that $\langle \theta, \sigma \rangle = f(H \leftarrow B)$ for some rule $H \leftarrow B$ in the domain of f.

For instance, the only rule of Π_1 applicable to $r(a)$ is

$$r(x) \leftarrow p(x), \mathit{not}\ q(x).$$

The function that maps this rule to $\langle \{x/y, y/x\}, \{y/a\} \rangle$ is a matching function for $r(a)$ and $\{x\}$ relative to Π_1. For this function f,

$$\mathit{Bodies}(f) = \{\langle \{p(y), \mathit{not}\ q(y)\}, \{y/a\} \rangle\}.$$

Two more pieces of notation will be needed. For any goal G, $\mathit{vars}(G)$ is the set of variables occurring in G. For any substitution δ and any set V of variables, $\delta \mid V$ is the restriction of δ to V:

$$x(\delta \mid V) = \begin{cases} x\delta, & \text{if } x \in V, \\ x, & \text{otherwise.} \end{cases}$$

Now we are ready to define the SLDNF calculus for a schematic normal program. It consists of the axiom

$$\models \emptyset : \epsilon$$

and the following inference rules:

$$(SP)\ \frac{\models (G \cup B)\sigma\ :\ \delta}{\models G \cup \{A\}\ :\ \sigma\delta \mid \mathit{vars}(G \cup \{A\})}$$

if $\langle B, \sigma \rangle \in \mathit{Bodies}(f)$,
where f is a matching function for A and $\mathit{vars}(G \cup \{A\})$

$$(FP)\ \frac{\dashv\models (G \cup B)\sigma \qquad \text{for all } \langle B, \sigma \rangle \in \mathit{Bodies}(f)}{\dashv\models G \cup \{A\}}$$

where f is a matching function for A and $\mathit{vars}(G \cup \{A\})$

$$(SN)\ \frac{\models G\ :\ \delta \qquad \dashv \{A\}}{\models G \cup \{not\ A\}\ :\ \delta} \qquad \text{if } A \text{ is ground}$$

$$(FN)\ \frac{\models \{A\}\ :\ \epsilon}{\dashv G \cup \{not\ A\}} \qquad \text{if } A \text{ is ground}$$

For example, here is a derivation in the SLDNF calculus for Π_1:

$$\cfrac{\cfrac{\cfrac{\models \emptyset\ :\ \epsilon \qquad (FP)\,\cfrac{}{\dashv q(a)}}{\models not\ q(a)\ :\ \epsilon}\,(SN)}{\models p(x), not\ q(x)\ :\ \{x/a\}}\,(SP)}{\models r(x)\ :\ \{x/a\}}\,(SP)$$

It is easy to check that the set of expressions derivable in this calculus is closed under the thinning rule from Section 2.3.

3.3 The Grounding Theorem

The grounding theorem tells us essentially that SLDNF is sound with respect to its propositional case.

Grounding Theorem. *For any schematic normal program Π, schematic goal G and substitution δ,*

- *if G succeeds relative to Π with c.a.s. δ then every ground instance of $G\delta$ succeeds relative to $Ground(\Pi)$,*
- *if G fails relative to Π then every ground instance of G fails relative to $Ground(\Pi)$.*

For instance, the derivation from Section 3.2 shows that $r(a)$ succeeds relative to $Ground(\Pi_1)$.

The soundness of SLDNF with respect to the answer set semantics for schematic programs is expressed by the following proposition:

Soundness Theorem. *For any schematic normal program Π, atomic formula A and substitution δ,*

- *if $\{A\}$ succeeds relative to Π with c.a.s. δ then every ground instance of $A\delta$ belongs to every answer set for Π,*
- *if $\{A\}$ fails relative to Π then no ground instance of A belongs to any answer set for Π.*

For instance, the derivation from Section 3.2 shows that $r(a)$ belongs to all answer sets for Π_1.

The soundness theorem is an immediate consequence of its propositional case (Section 2.2) and of the grounding theorem stated above.

3.4 Prolog

The "evaluation" of a schematic goal G is the process of either generating a series of computed answer substitutions for G or establishing that G fails. Prolog — assuming that its unification algorithm produces correct results[5] — attempts to evaluate a given goal G by backward chaining. The first step, as in the propositional case (Section 2.3), is to select a singleton subgoal in G.

If the selected subgoal does not contain negation as failure, so that G is represented as $G' \cup \{A\}$, then Prolog selects a matching function f for A and $vars(G' \cup \{A\})$ and generates, one by one, all elements $\langle B, \sigma \rangle$ of $Bodies(f)$. For each of them, it starts evaluating $(G' \cup B)\sigma$. Whenever an expression of the form $\models (G' \cup B)\sigma : \delta$ is derived, a new c.a.s. for $G' \cup \{A\}$ is produced by one application of (SP). If $\dashv (G' \cup B)\sigma$ is derived for each pair $\langle B, \sigma \rangle$ then the failure of $G' \cup \{A\}$ is concluded by one application of (FP).

If the selected subgoal does contain negation as failure, so that G is represented as $G' \cup \{not\ A\}$, and A is a ground atom, then the rule applied in the derivation last will be (SN), (FN) or (T), as in the propositional case. If A is nonground then Prolog "flounders." It is not capable of continuing the evaluation process.

4 Constructive Negation

4.1 SLDNF with Constructive Negation

We turn now to an extension of the calculus from Section 3.2 that corresponds to the use of constructive negation. In the new calculus, the derivable objects are $\models G : \Gamma$ and $\dashv G : \Gamma$, where G is a schematic goal and Γ is a set of grounding substitutions.[6] If $\models G : \Gamma$ is derivable then we say that G *succeeds on* Γ relative to Π; if $\dashv G : \Gamma$ is derivable then we say that G *fails outside of* Γ relative to Π. The general form of the grounding theorem will be stated as follows: if G succeeds on Γ relative to Π then, for every grounding substitution $\gamma \in \Gamma$, $G\gamma$ succeeds relative to $Ground(\Pi)$; if G fails outside of Γ relative to Π then, for every grounding substitution $\gamma \notin \Gamma$, $G\gamma$ fails relative to $Ground(\Pi)$. The expression $\models G : \delta$ of the calculus from Section 3.2 corresponds, in the new notation, to $\models G : \Gamma$ where Γ is the set of ground instances of δ; $\dashv G$ corresponds to $\dashv G : \emptyset$.

By Γ_0 we denote the set of grounding substitutions. For any substitution δ and any subset Γ of Γ_0, $\delta\Gamma$ stands for $\{\delta\gamma : \gamma \in \Gamma\}$. In particular, $\delta\Gamma_0$ is the set of ground instances of δ. For any subset Γ of Γ_0 and any set V of variables, define

$$\Gamma \mid V = \{\gamma \in \Gamma_0 : \text{for some } \gamma' \in \Gamma, \gamma \mid V = \gamma' \mid V\}.$$

The axioms of the SLDNF calculus with constructive negation are the expressions of the forms

$$\models \emptyset : \Gamma_0, \quad \,=\!\!\!\mid G : \Gamma_0.$$

The inference rules[7] are:

$$(SP_c)\ \frac{\models (G \cup B)\sigma \ : \ \Gamma}{\models G \cup \{A\} \ : \ \sigma\Gamma \mid \mathit{vars}(G \cup \{A\})}$$

if $\langle B, \sigma \rangle \in \mathit{Bodies}(f)$,
where f is a matching function for A and $\mathit{vars}(G \cup \{A\})$

$$(FP_c)\ \frac{=\!\!\!\mid (G \cup B)\sigma \ : \ \Gamma_{B,\sigma} \qquad \text{for all } \langle B, \sigma \rangle \in \mathit{Bodies}(f)}{=\!\!\!\mid G \cup \{A\} \ : \ \left(\bigcup_{\langle B,\sigma\rangle \in \mathit{Bodies}(f)} \sigma\Gamma_{B,\sigma}\right) \mid \mathit{vars}(G \cup \{A\})}$$

where f is a matching function for A and $\mathit{vars}(G \cup \{A\})$

$$(SN_c)\ \frac{\models G \ : \ \Gamma \qquad =\!\!\!\mid \{A\} \ : \ \Gamma'}{\models G \cup \{\mathit{not}\ A\} \ : \ \Gamma \setminus \Gamma'}$$

$$(FN_c)\ \frac{=\!\!\!\mid G \ : \ \Gamma \qquad \models \{A\} \ : \ \Gamma'}{=\!\!\!\mid G \cup \{\mathit{not}\ A\} \ : \ \Gamma \setminus \Gamma'}$$

Here, for instance, is the derivation in the new calculus for program Π_1 that corresponds to the example from Section 3.2:

$$(SP_c)\ \cfrac{(SP_c)\ \cfrac{(SN_c)\ \cfrac{\models \emptyset \ : \ \Gamma_0 \qquad (FP_c)\ \cfrac{}{=\!\!\!\mid q(a) \ : \ \emptyset}}{\models \mathit{not}\ q(a) \ : \ \Gamma_0}}{\models p(x), \mathit{not}\ q(x) \ : \ \{x/a\}\Gamma_0}}{\models r(x) \ : \ \{x/a\}\Gamma_0}$$

Thus $r(x)$ succeeds on $\{x/a\}\Gamma_0$. This example illustrates the general theorem which shows that the new calculus is at least as powerful as the old one:

Embedding Theorem. *For any schematic normal program Π, schematic goal G and substitution δ,*

- *if G succeeds relative to Π with c.a.s. δ then G succeeds relative to Π on a superset of $\delta\Gamma_0$,*

- *if G fails relative to Π then G fails relative to Π outside of $\emptyset$.*

On the other hand, the inference rules of the extended calculus sometimes allow us to arrive at conclusions that have no counterparts in the calculus of Section 3.2. Replace, for instance, the last rule of Π_1

$$r(x) \leftarrow p(x), \mathit{not}\ q(x)$$

by

$$r(x) \leftarrow \mathit{not}\ q(x).$$

For the new schematic program Π_2, the goal $r(x)$ neither succeeds nor fails; on this goal, Prolog would flounder (Section 3.4.) With constructive negation, the situation is different: $r(x)$ succeeds on $\{x/a\}\Gamma_0$ with respect to Π_2, just as it does with respect to Π_1. Indeed, consider the following derivation in the SLDNF calculus with constructive negation for Π_2:

$$(SP_c)\ \dfrac{(SN_c)\ \dfrac{\models \emptyset\ :\ \Gamma_0 \qquad (FP_c)\ \dfrac{=\!\!|\ \emptyset\ :\ \Gamma_0}{=\!\!|\ q(x)\ :\ \{x/b\}\Gamma_0}}{\models \mathit{not}\ q(x)\ :\ \Gamma_0 \setminus \{x/b\}\Gamma_0}}{\models r(x)\ :\ \Gamma_0 \setminus \{x/b\}\Gamma_0}$$

It remains to notice that

$$\Gamma_0 \setminus \{x/b\}\Gamma_0 = \{x/a\}\Gamma_0,$$

because a and b are the only ground terms in the language of Π_2.

4.2 The Grounding Theorem for Constructive Negation

The embedding theorem shows that the calculus of Section 4.1 is at least as strong as the SLDNF calculus. The grounding theorem stated below shows, on the other hand, that it is not too strong.

Grounding Theorem for Constructive Negation. *For any schematic normal program Π, schematic goal G and subset Γ of Γ_0,*

- *if G succeeds on Γ relative to Π then, for every $\gamma \in \Gamma$, $G\gamma$ succeeds relative to $Ground(\Pi)$,*
- *if G fails outside of Γ relative to Π then, for every $\gamma \in \Gamma_0 \setminus \Gamma$, $G\gamma$ fails relative to $Ground(\Pi)$.*

This fact, combined with the embedding theorem, implies the grounding theorem from Section 3.3. In combination with the soundness theorem for propositional SLDNF (Section 2.2), it leads to the following corollary:

Soundness Theorem for Constructive Negation. *For any schematic normal program Π, atomic formula A and subset Γ of Γ_0,*

- *if $\{A\}$ succeeds on Γ relative to Π then, for every $\gamma \in \Gamma$, $A\gamma$ belongs to every answer set for Π,*
- *if $\{A\}$ fails outside of Γ relative to Π then, for every $\gamma \in \Gamma_0 \setminus \Gamma$, $A\gamma$ does not belong to any answer set for Π.*

4.3 Prolog with Constructive Negation

It would be interesting to investigate whether the algorithms used in the existing systems of Prolog with constructive negation can be described as proof procedures for the SLDNF calculus with constructive negation or for its extensions, in the spirit of Sections 2.3 and 3.4. This has not been attempted in any detail. The following comments can be relevant.

Derivations in the calculus from Section 4.1 will more closely resemble the examples found in the literature on constructive negation if we represent the sets of grounding substitutions that occur in these derivations by formulas. Recall that the language of every schematic program is a first-order language $\mathbf{L}$ without equality (Section 3.1). Consider the first-order language $\mathbf{L}^=$ with equality that has the same object and function constants as $\mathbf{L}$, and that has no predicate constants other than equality. For any formula F of $\mathbf{L}^=$, let $\mathrm{Set}[F]$ be the set of the grounding substitutions γ for which the sentence $F\gamma$ is true in the Herbrand interpretation of $\mathbf{L}^=$. For instance,

$$\mathrm{Set}[x = a] = \{x/a\}\Gamma_0.$$

If a schematic program Π is finite, then any set Γ of grounding substitutions that can appear in a derivation in the SLDNF calculus with constructive negation for Π can be represented in the form $\mathrm{Set}[F]$. For any formula F of $\mathbf{L}^=$ and any atomic formulas $A_1, \ldots, A_n$ of $\mathbf{L}$, let

$$\models F, A_1, \ldots, A_n$$

stand for

$$\models \{A_1, \ldots, A_n\} \; : \; \mathrm{Set}[F],$$

and let

$$=\!\mid F, A_1, \ldots, A_n$$

stand for

$$\models \{A_1, \ldots, A_n\} \; : \; \Gamma_0 \setminus \mathrm{Set}[F].$$

In this notation, the last example of Section 4.1 can be written as follows:

$$\cfrac{\models true \qquad (FP_c)\cfrac{=\!\!| \, false}{=\!\!| \, x \neq b, q(x)}}{\cfrac{\models x \neq b, not\ q(x)}{\models x \neq b, r(x)}(SP_c)}(SN_c)$$

With derivations written in this format, the calculus from Section 4.1 seems to be particularly close to the methods of handling constructive negation proposed in [7].

SLD-CNF resolution, introduced by Chan [5], uses both formulas of $\mathbf{L}^=$ and computed answer substitutions. The success of $\leftarrow F, A_1, \ldots, A_n$ under SLD-CNF resolution with c.a.s. σ is expressed, in our notation, by

$$\models \{A_1, \ldots, A_n\} \ : \ \sigma \mathrm{Set}[F].$$

Acknowledgements

I would like to thank Włodzimierz Drabent and Norman McCain for comments on drafts of this paper. I have also benefited from discussing the problems related to the subject of this paper with Krzysztof Apt, Nicole Bidoit, Michael Gelfond, Neelakantan Kartha, Teodor Przymusinski, Raymond Reiter, Grigori Schwarz, Robert Stärk and Hudson Turner. This work was partially supported by National Science Foundation under grant IRI-9306751.

Notes

1. Similar ideas were developed earlier by Mints [16] in the context of pure Prolog. That work was extended in [4], [20] and [21]. Some of the other papers that anticipated, expounded and developed Kunen's approach to SLDNF are [1], [2], [9], [19] and [22]. In [14], these ideas are applied to the study of loop checking.

2. Recall that, for a calculus with infinitary rules, a derivation is defined as a tree without infinite branches, whose nodes are derivable objects such that every node either (a) is a leaf and an axiom, or (b) can be obtained by applying an inference rule to its successor nodes.

3. For the definitions related to the notion of a substitution, see [8], Sections 5.2 and 7.2.

4. A substitution θ is *invertible* if there is a substitution θ^{-1} such that both $\theta\theta^{-1}$ and $\theta^{-1}\theta$ equal the identity substitution ϵ.

5. This qualification is needed in view of the fact that most Prolog systems omit the occurs check from the unification algorithm.

6. A substitution is *grounding* if it maps every variable of the language to a ground term.

7. The rule (FN_c) below was suggested to me by Włodzimierz Drabent.

References

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

[2] Krzysztof Apt and Kees Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18:177–190, 1994.

[3] Nicole Bidoit and Christine Froidevaux. Minimalism subsumes default logic and circumscription. In *Proc. of LICS-87*, pages 89–97, 1987.

[4] Serenella Cerrito. A linear axiomatization of negation as failure. *Journal of Logic Programming*, 12(1–2):1–24, 1992.

[5] David Chan. Constructive negation based on the completed database. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 111–125, 1988.

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

[7] Włodzimierz Drabent. What is failure? An approach to constructive negation. *Acta Informatica*, 32(1):27–59, 1995.

[8] Melvin Fitting. *First-order logic and automated theorem proving.* Springer-Verlag, 1990.

[9] Dov Gabbay and Marek Sergot. Negation as inconsistency. *Journal of Logic Programming*, 3(1):1–36, 1986.

[10] Michael Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.

[11] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

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

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

[14] Vladimir Lifschitz, Norman McCain, Teodor Przymusinski, and Robert Stärk. A calculus of goals for loop checking. In *Logic Programming and Non-monotonic Reasoning: Proceedings of the Third International Conference*, 1995. To appear.

[15] John Lloyd. *Foundations of logic programming.* Springer-Verlag, 1987. Second, extended edition.

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

[17] Robert Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[18] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81–132, 1980.

[19] John Shepherdson. Logics for negation as failure. In Yiannis Moschovakis, editor, *Logic from Computer Science*, pages 521–583. Springer-Verlag, 1992.

[20] John Shepherdson. Mints type deductive calculi for logic programming. *Annals of Pure and Applied Logic*, 56:7–17, 1992.

[21] Robert Stärk. The declarative semantics of the Prolog selection rule. In *Proc. Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 252–261, 1994.

[22] Robert Stärk. Input/output dependencies of normal logic programs. *Journal of Logic and Computation*, 4(3):249–262, 1994.