Nested Expressions in Logic Programs (extended abstract)

Vladimir Lifschitz, Lappoon R. Tang and Hudson Turner
University of Texas at Austin
Austin, TX 78712, USA
{vl,rupert,hudson}@cs.utexas.edu

> As an example, if `not` were declared as a prefix operator of type `fy`, then
>
> `not not P`
>
> would be a permissible way to write `not(not(P))`.
>
> (From the *SICStus Prolog Manual*)

Abstract

We extend the answer set semantics to a class of programs with nested expressions permitted in the bodies and heads of rules. These expressions are formed from literals using negation as failure, conjunction (,) and disjunction (;) that can be nested arbitrarily. Conditional expressions are introduced as abbreviations. The study of equivalent transformations of programs with nested expressions shows that any such program is equivalent to a set of disjunctive rules, possibly with negation as failure in the heads. The generalized answer set semantics is related to the Lloyd-Topor generalization of Clark's completion and to the logic of minimal belief and negation as failure.

1 Introduction

In most papers on the semantics of logic programming, the body of a rule is supposed to be a list of syntactically simple expressions, such as atoms or literals. The syntax of Prolog, however, permits nested expressions, and the use of nesting does not appear to make a program any less declarative. If we know how to give a declarative meaning to rules like

$$p \leftarrow q, \mathit{not}\, r$$

then why don't we allow

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

as well? And while we are at it, we can also include disjunctions and conditional expressions, and thus extend the semantics to rules like

$$p \leftarrow q; \mathit{not}\, r$$

and

$$p \leftarrow (q \rightarrow r; s)\,.$$

In this note we propose an extension of the answer set semantics [4] to a class of programs with nested expressions permitted in the rules. These expressions are formed from literals (atoms possibly preceded by the classical negation sign $\neg$) using the operators

$$,\quad ;\quad \mathit{not} \tag{1}$$

that can be nested arbitrarily. Conditional expressions will be introduced as abbreviations.

An interesting feature of the generalization defined in this paper is that two negation as failure operators in a row do not, generally, cancel each other. Our definition, applied, for instance, to the program

$$p \leftarrow \mathit{not}\, \mathit{not}\, p\,, \tag{2}$$

gives both $\emptyset$ and $\{p\}$ as the answer sets, although

$$p \leftarrow p$$

has only one answer set $\emptyset$. The following informal argument shows that this is in fact reasonable. In (2), we can "denote" the subformula $not\, p$ by q, and (2) will turn into

$$
\begin{aligned}
p &\leftarrow not\, q\,, \\
q &\leftarrow not\, p\,.
\end{aligned}
$$

The answer sets for this program are $\{p\}$ and $\{q\}$. After dropping the "auxiliary" symbol q from each of them, we will get the two answer sets shown above.

The syntax and semantics of programs with nested expressions are defined in Sections 2 and 3. The theorem stated in Section 4 shows that any such program can be equivalently transformed into a set of "disjunctive rules" in the sense of Section 5.1 of [8]—expressions of the form

$$L_1; \ldots; L_k; not\ L_{k+1}; \ldots; not\ L_l \leftarrow L_{l+1}, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n \quad (3)$$

where $0 \leq k \leq l \leq m \leq n$ and all L_i are literals.[1] A somewhat unorthodox feature of (3) is the possibility of negation as failure in the head.[2] This possibility is essential for us here, because in the process of "unwinding" a nested expression in the body some occurrences of negation as failure can move to the head.

In Section 5, we study the relation of our proposal to the system of [11], which extends the completion semantics of [2] to programs with arbitrary first-order formulas in the bodies of rules.

Several authors, including Lin and Shoham [10], Lifschitz [7], Herre and Wagner [5] and Pearce [12], described ways to embed logic programs under the answer set (stable model) semantics into languages that are closed under the formation rules of propositional logic. In Section 6, the embedding given in the paper by Lifschitz is extended to logic programs with nested expressions.

2 Syntax

The words *atom* and *literal* are understood here as in propositional logic. To extend this semantics to programs with variables, we make these programs

[1]In [8], the vertical bar is used instead of the semicolon to separate literals in the head of a rule.

[2]This feature was found to be useful in abductive logic programming [6].

propositional by grounding.

Elementary formulas are literals and the symbols $\bot$ (“false”) and $\top$ (“true”). *Formulas* are built from elementary formulas using the unary connective *not* and the binary connectives , (conjunction) and ; (disjunction). A *rule* is an expression of the form

$$F \leftarrow G$$

where F and G are formulas, called the *head* and the *body* of the rule.

This definition of a rule generalizes the syntax of rules in the sense of [4], and even of “disjunctive rules” (3). (We understand the head of (3) as $\bot$ if $l = 0$, and understand the body as $\top$ if $n = l$.)

For any formulas F, G and H,

$$F \rightarrow G; H$$

stands for the formula

$$(F, G); (not\, F, H)\,.$$

We will identify a rule of the form $F \leftarrow \top$ with the formula F.

A *program* is a set of rules.

Note that, syntactically, the status of the classical negation operator $\neg$ is different from the status of operators (1): classical negation is allowed only in front of an atom. About an occurrence of a formula F in a formula or a rule we will say that it is *singular* if the symbol before this occurrence is $\neg$; otherwise the occurrence is *regular*. It is clear that an occurrence of F can be singular only if F is an atom. For instance, in the formula

$$not\, p, not\, \neg p \tag{4}$$

the first occurrence of p is regular, and the second is singular. This difference is important because the result of replacing a regular occurrence of F in a formula or a rule by another formula G is again a formula or a rule, but for a singular occurrence this is not necessarily the case. For instance, the result of replacing the first occurrence of p in (4) by (q, r) is a formula, but the result of applying the same operation to the second occurrence is not.

3 Semantics

Recall that the definition of an answer set [4] consists of two parts. First this concept is defined for the "basic" case of programs that do not contain negation as failure. Then the "reduct" of a program relative to a set of literals is defined, and this is used to reduce the general case to the basic case. The extension of this definition to programs in the sense of Section 2 follows the same plan.

The formulas, rules and programs that do not contain the negation as failure operator *not* will be called *basic*. We define when a set X of literals *satisfies* a basic formula F (symbolically, $X \models F$) recursively, as follows:

- for elementary F, $X \models F$ if X entails F in classical propositional logic (that is, if $F \in X$, or $F = \top$, or X contains a complementary pair).
- $X \models (F, G)$ if $X \models F$ and $X \models G$.
- $X \models (F; G)$ if $X \models F$ or $X \models G$.

Let Π be a basic program. A set X of literals is *closed* under Π if, for every rule $F \leftarrow G$ in Π, $X \models F$ whenever $X \models G$. We say that X is an *answer set* for Π if X is a minimal set of literals that is both logically closed[3] and closed under Π.

Example 1. Consider the program whose only rule is

$$q \leftarrow p; \neg p\,. \tag{5}$$

Being closed under (5) is characterized by the following condition: if $p \in X$ or $\neg p \in X$ then $q \in X$. It is clear that the only answer set for (5) is empty. If we add the rule p (that is, $p \leftarrow \top$) to this program then $\{p, q\}$ will be the only answer set.

The *reduct* of a formula, rule or program relative to a set X of literals is defined recursively, as follows:

- for elementary F, $F^X = F$.
- $(F, G)^X = (F^X, G^X)$.

[3] A set of literals is said to be *logically closed* if it is consistent or contains all literals.

- $(F; G)^X = (F^X; G^X)$.
- $(not\ F)^X = \begin{cases} \bot, & \text{if } X \models F^X, \\ \top, & \text{otherwise.} \end{cases}$
- $(F \leftarrow G)^X = F^X \leftarrow G^X$.
- $\Pi^X = \{(F \leftarrow G)^X : F \leftarrow G \in \Pi\}$.

A set X of literals is an *answer set* for Π if it is an answer set for the reduct Π^X.

Example 2. Consider the program whose only rule is

$$p \leftarrow (q \rightarrow r; not\ s), \tag{6}$$

that is,

$$p \leftarrow (q, r); (not\ q, not\ s).$$

Take $X = \{p\}$; let us check that X is an answer set for this program. Note that $q^X = q$, so that $q^X \notin X$ and $X \not\models q^X$. Consequently $(not\ q)^X = \top$. Similarly, $(not\ s)^X = \top$. It follows that

$$((q, r); (not\ q, not\ s))^X = (q, r); (\top, \top).$$

Consequently the reduct of (6) relative to X is

$$p \leftarrow (q, r); (\top, \top).$$

The only answer set for this program is $\{p\}$, so that this is indeed an answer set for (6). (There are no other answer sets.)

Example 3. Consider program (2). For $X_1 = \emptyset$, we note that $p^{X_1} \notin X_1$, which means that $X_1 \not\models p^{X_1}$ and consequently $(not\ p)^{X_1} = \top$. It follows that $X_1 \models (not\ p)^{X_1}$, so that $(not\ not\ p)^{X_1} = \bot$. The reduct of (2) relative to X_1 is

$$p \leftarrow \bot.$$

The only answer set for this program is empty, so that X_1 is indeed an answer set for (2). Now take $X_2 = \{p\}$. Note that $p^{X_2} \in X_2$, so $X_2 \models$

p^{X_2}. Consequently $(not\, p)^{X_2} = \bot$. It follows that $X_2 \not\models (not\, p)^{X_2}$, so that $(not\; not\, p)^{X_2} = \top$. The reduct of (2) relative to X_2 is

$$p \leftarrow \top\,.$$

The only answer set for this program is $\{p\}$, so that X_2 is an answer set for (2) also.

Proposition 1. *For a program whose rules have the form (3), the definition of an answer set given above is equivalent to the definition from [8].*

4 Normal Form

We say that a program Π_1 is *equivalent* to a program Π_2 if, for any sets X and Y of literals, X is closed under Π_1^Y iff X is closed under Π_2^Y. It is clear that the reducts of two equivalent programs relative to the same set of literals have the same answer sets. Consequently, any two equivalent programs have the same answer sets.

Proposition 2. *Any program is equivalent to a set of rules of form (3).*

Examination of the proof of Proposition 2 shows that the rules of the form (3) generated from a program Π do not contain negation as failure in the heads ($k = l$) if negation as failure in Π

(i) does not occur in the heads of rules, and

(ii) is not nested, that is, not applied to formulas containing negation as failure.

The answer sets for any collection of rules of the form (3) without negation as failure in the heads are known to have the anti-chain property: an answer set for such a program cannot be a proper subset of another answer set for the same program. It follows that the answer sets for any collection of rules satisfying conditions (i) and (ii) form an anti-chain. Program (2) does not satisfy condition (ii), and one of its two answer sets is a subset of the other.

5 Relation to the Lloyd-Topor Semantics

The review of the Lloyd-Topor semantics below differs from its description in [11] in two ways. First, it is restricted to the propositional case. Second, it is stated in terms of "supported models" [1] instead of completion, which allows us to extend the theory to infinite programs. For finite programs, the two definitions are equivalent.

A *(propositional) Lloyd-Topor rule* is an expression of the form

$$p \leftarrow F$$

where p is an atom and F is a propositional formula. A *Lloyd-Topor program* is a set of Lloyd-Topor rules.

Recall that a *(propositional) interpretation* is a function from atoms to truth values. We will identify an interpetation with the set of atoms to which it assigns the value *true*.

Let Π be a Lloyd-Topor program. A *model* of Π is an interpretation I such that, for any rule $p \leftarrow F$ in Π such that I satisfies F, I satisfies p also. A model I of Π is *supported* if, for any atom p such that I satisfies p, there is a rule $p \leftarrow F$ in Π such that I satisfies F.

For instance, the Lloyd-Topor program whose only rule is $p \leftarrow p$ has two supported models: $\emptyset$ and $\{p\}$.

The translation τ of propositional formulas, and of Lloyd-Topor rules and programs, to our language is defined as follows, assuming that all connectives other than $\wedge$, $\vee$ and $\neg$ have been eliminated:

- for atomic F, $\tau F = \mathit{not}\ \mathit{not}\ F$.
- $\tau(F \wedge G) = \tau F, \tau G$.
- $\tau(F \vee G) = \tau F; \tau G$.
- $\tau(\neg F) = \mathit{not}\ \tau F$.
- $\tau(p \leftarrow F) = p \leftarrow \tau F$.
- $\tau\Pi = \{\tau(p \leftarrow F) : p \leftarrow F \in \Pi\}$.

The soundness of this translation is expressed by the following theorem:

Proposition 3. *For any Lloyd-Topor program* Π *and any interpretation* I, I *is a supported model of* Π *iff* I *is an answer set for* $\tau\Pi$.

Since the heads of the rules of $\tau\Pi$ are atoms, any answer set for this program is a set of atoms, that is to say, an interpretation. Consequently, we can conclude from Proposition 3 that every answer set for $\tau\Pi$ is a supported model of Π.

For instance, $\tau(p \leftarrow p)$ is (2). In accordance with Proposition 3, the two answer sets for (2) computed in Example 3 are identical to the two supported models of $p \leftarrow p$. As another example, the result of applying τ to

$$p \leftarrow \neg q \tag{7}$$

is

$$p \leftarrow not\ not\ not\ q\,.$$

This program is equivalent to

$$p \leftarrow not\ q\,.$$

We see that, in application to (7), the Lloyd-Topor semantics is equivalent to the answer set semantics, modulo replacing $\neg$ by *not*. This is not surprising in view of the fact that the result of this replacement is a tight program (see [8], and the result from [3] reproduced there as Proposition 3.5).

6 Relation to MBNF

In the propositional fragment of the logic of minimal belief and negation as failure (MBNF) [7], formulas are built from atoms using propositional connectives and two modal operators: B ("minimal belief") and *not*. An *MBNF theory* is a set of formulas (axioms) in this language. We will introduce a translation μ that maps formulas and rules in the sense of Section 2 to formulas of MBNF, and maps programs to MBNF theories. The translation is defined recursively, as follows:

- for elementary F, $\mu F = \mathsf{B}F$.
- $\mu(F, G) = \mu F \wedge \mu G$.
- $\mu(F; G) = \mu F \vee \mu G$.

- for elementary F, $\mu(not\, F) = not\, F$.
- $\mu(not\,(F, G)) = \mu(not\, F) \vee \mu(not\, G)$.
- $\mu(not\,(F; G)) = \mu(not\, F) \wedge \mu(not\, G)$.
- $\mu(not\, not\, F) = \neg\mu(not\, F) \wedge not\, \bot$.
- $\mu(F \leftarrow G) = \mu G \supset \mu F$.
- $\mu\Pi = \{\mu(F \leftarrow G) : F \leftarrow G \in \Pi\}$.

As an example, note that, in application to (2), μ gives the formula

$$\neg not\, p \wedge not\, \bot \supset \mathsf{B}p\,.$$

This translation is essentially an extension of the translation from Section 5 of [7] to programs with nested expressions. The translation $\mu\Pi$ of any program Π is an MBNF theory of the special type studied in [9]—a "theory with protected literals." This means that every occurrence of an atom in an axiom of this theory is a part of an expression of the form $\mathsf{B}L$ or $not\, L$, where L is a literal.

Recall that *models* of a propositional MBNF theory are defined as the pairs (I, S), where I is an interpretation and S is a set of interpretations, that satisfy a certain fixpoint condition ([7], Section 4).

The following theorem establishes a correspondence between the answer sets for a program Π and the models of the MBNF theory $\mu\Pi$. For any set X of literals, by $\mathrm{Mod}(X)$ we denote the set of interpretations that satisfy all members of X.

Proposition 4. *For any program Π, logically closed set X of literals, and interpretation I, X is an answer set for Π iff $(I, \mathrm{Mod}(X))$ is a model of $\mu\Pi$. Moreover, every model of $\mu\Pi$ can be represented in the form $(I, \mathrm{Mod}(X))$, where I is an interpretation and X is a logically closed set of literals.*

Acknowledgements

We are grateful to Esra Erdem, Michael Gelfond, David Pearce, Phoebe Weidmann and David S. Warren for useful discussions on the subject of this paper. This work was partially supported by National Science Foundation under grant IRI-9306751.

References

[1] Krzysztof Apt, Howard Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, San Mateo, CA, 1988.

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

[3] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

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

[5] Heinrich Herre and Gerd Wagner. Stable models are generated by a stable chain. *Journal of Logic Programming*, 30:165–177, 1997.

[6] Katsumi Inoue and Chiaki Sakama. On positive occurrences of negation as failure. In *Proc. Fourth Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 293–304, 1994.

[7] Vladimir Lifschitz. Minimal belief and negation as failure. *Artificial Intelligence*, 70:53–72, 1994.

[8] Vladimir Lifschitz. Foundations of logic programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 69–128. CSLI Publications, 1996.

[9] Vladimir Lifschitz and Thomas Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 603–614, 1992.

[10] Fangzhen Lin and Yoav Shoham. A logic of knowledge and justified assumptions. *Artificial Intelligence*, 57:271–289, 1992.

[11] John Lloyd and Rodney Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.

[12] David Pearce. A new logical characterization of stable models and answer sets. In Jürgen Dix, Luis Pereira, and Teodor Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming (Lecture Notes in Artificial Intelligence 1216)*, pages 57–70. Springer-Verlag, 1997.