Datalog Programs and Their Stable Models

Vladimir Lifschitz

Department of Computer Science
University of Texas at Austin, USA

Abstract. This paper is about the functionality of software systems used in answer set programming (ASP). ASP languages are viewed here, in the spirit of Datalog, as mechanisms for characterizing intensional (output) predicates in terms of extensional (input) predicates. Our approach to the semantics of ASP programs is based on the concept of a stable model defined in terms of a modification of parallel circumscription.

1 Introduction

This paper is about the functionality of software systems used in answer set programming (ASP) [11, 14, 1, 7]. ASP languages are viewed here, in the spirit of Datalog, as mechanisms for characterizing intensional (output) predicates in terms of extensional (input) predicates.

Example 1. The ASP program

```
q(X,Y) :- p(X,Y).
q(X,Z) :- q(X,Y), q(Y,Z).
```

can be viewed as a definition of the output predicate q in terms of the input predicate p; it tells us that q is the transitive closure of p. To illustrate this assertion, consider what happens when we extend the program above by a set of ground atoms defining p, such as

```
p(a,b). p(b,c).
```

Given the file consisting of these three lines, an ASP system such as CLINGO[1] or DLV[2] returns the transitive closure of p:[3]

```
{q(a,b), q(a,c), q(b,c)}.
```

Example 2. Take the disjunctive ASP program consisting of one rule

```
q(X) ; r(X) :- p(X).
```

[1] http://potassco.sourceforge.net

[2] http://www.dlvsystem.com

[3] To be precise, the set of atoms generated by these systems includes also the atoms defining p.

It can be thought of as a description of all possible ways to partition an input p into disjoint[4] (and possibly empty) subsets q, r. Consider, for instance, what happens when we combine this rule with a set of ground atoms defining p, such as

```
p(a). p(b). p(c).
```

Given this file, DLV returns the list of 8 partitions:

```
{r(a), r(b), r(c)},
{q(a), r(b), r(c)},
{r(a), q(b), r(c)},
{q(a), q(b), r(c)},
{r(a), r(b), q(c)},
{q(a), r(b), q(c)},
{r(a), q(b), q(c)},
{q(a), q(b), q(c)}.
```

Example 3. The choice rule

```
{q(X)} :- p(X).
```

describes all possible ways to choose a subset q of a given set p. Given this one-rule program and the same input as in Example 2, CLINGO generates all subsets of $\{a, b, c\}$:

```
{ },
{q(a)},
{q(b)},
{q(b), q(a)},
{q(c)},
{q(c), q(a)},
{q(c), q(b)},
{q(c), q(b), q(a)}.
```

We describe here a declarative semantics for a class of ASP programs that includes many examples of this kind. Our approach is based on the concept of a stable model [5] generalized as proposed in [3]. We will see, for instance, that the stable models of the program from Example 1 are arbitrary interpretations (in the sense of first-order logic) of the language with binary predicate constants p, q in which q is the transitive closure of p. The stable models of the program from Example 3 are arbitrary interpretations of the language with unary predicate constants p, q in which q is a subset of p.

[4] Disjunction in the head of an ASP rule often behaves as exclusive disjunction, but there are exceptions. See the discussion of this example in Section 4.

2 A Few More Examples

We will now extend the program from Example 2 by adding a "constraint"—a rule with the empty head. The effect of adding a constraint to an ASP program is to weed out the solutions satisfying the body of the constraint.

Example 4. The program

```
q(X) ; r(X) :- p(X).
:- q(a).
```

describes the partitions of the input p into subsets q, r such that a is not in q. Given this program and the same input as in Example 2, DLV returns

```
{r(a), r(b), r(c)},
{r(a), q(b), r(c)},
{r(a), r(b), q(c)},
{r(a), q(b), q(c)}.
```

Example 5. If p is the set of vertices of a directed graph, and q is the set of its edges, then the program

```
r(X) :- q(X,Y).
s(X) :- p(X), not r(X).
```

describes the set s of terminal vertices. It uses the auxiliary symbol r, representing the complement of s. Given this program and the input

```
p(a). p(b). q(a,b).
```

both CLINGO and DLV return

```
{r(a), s(b)}.
```

The combination

```
not r(X)
```

in the body of the second rule employs "negation as failure" to express that the rules of the program do not allow us to establish `r(X)`. In Section 5 we will see how the stable model semantics makes this idea precise.

Example 6. For p and q as in the previous example, the program below defines the sets of vertices of out-degrees 0, 1, and 2:

```
r0(X) :- p(X), #count{Y:q(X,Y)}=0.
r1(X) :- p(X), #count{Y:q(X,Y)}=1.
r2(X) :- p(X), #count{Y:q(X,Y)}=2.
```

In particular, r_0 has the same meaning as s from Example 5. The "aggregate" symbol `#count` used in these rules represents the cardinality of a set. Given this program and the input

```
p(a). p(b). p(c). q(a,b). q(a,c).
```

DLV returns

```
{r0(b), r0(c), r2(a)}.
```

	Logic programming notation	First-order formula
1	`q(X,Y) :- p(X,Y).`	$\forall xy(p(x,y) \rightarrow q(x,y))$
2	`q(X,Z) :- q(X,Y), q(Y,Z).`	$\forall xyz(q(x,y) \wedge q(y,z) \rightarrow q(x,z))$
3	`q(X) ; r(X) :- p(X).`	$\forall x(p(x) \rightarrow q(x) \vee r(x))$
4	`{q(X)} :- p(X).`	$\forall x(p(x) \rightarrow q(x) \vee \neg q(x))$
5	`:- q(a).`	$q(a) \rightarrow \bot$
6	`r(X) :- q(X,Y).`	$\forall xy(q(x,y) \rightarrow r(x))$
7	`s(X) :- p(X), not r(X).`	$\forall x(p(x) \wedge \neg r(x) \rightarrow s(x))$
8	`r0(X) :- p(X), #count{Y:q(X,Y)}=0.`	$\forall x(p(x) \wedge \neg(\exists y)q(x,y) \rightarrow r_0(x))$
9	`r1(X) :- p(X), #count{Y:q(X,Y)}=1.`	$\forall x(p(x) \wedge (\exists y)q(x,y) \wedge \neg(\exists_2 y)q(x,y) \rightarrow r_1(x))$
10	`r2(X) :- p(X), #count{Y:q(X,Y)}=2.`	$\forall x(p(x) \wedge (\exists_2 y)q(x,y) \wedge \neg(\exists_3 y)q(x,y) \rightarrow r_2(x))$

Fig. 1. Rules as formulas

3 Rules and Programs

In first-order formulas, we take the symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$ to be primitives, along with the 0-place connectives $\top$ (truth) and $\bot$ (falsity).

A first-order sentence is a *rule* if it has the form

$$\widetilde{\forall}(B \rightarrow H) \tag{1}$$

and has no occurrences of $\rightarrow$ other than the one explicitly shown.[5] Formula B is the *body* of rule (1), and H is its *head*. The expressions that were called rules in Examples 1–6 can be viewed as rules in the sense of this definition written in "logic programming notation," as shown in Figure 1. In the last two lines, we use the abbreviation $\exists_n x F(x)$ for

$$\exists x_1 \cdots x_n \left(\bigwedge_{1 \le i \le n} F(x_i) \wedge \bigwedge_{1 \le i < j \le n} x_i \neq x_j \right).$$

Note that $\neg r(x)$ in line 7 of the table corresponds to `not r(X)` in logic programming notation. When we write a rule as a formula, the negation symbol $\neg$ corresponds to negation as failure, and not to "classical" (or "strong") negation in the sense of [6]. (To represent rules containing strong negation as first-order formulas, we would have to eliminate strong negation from them in favor of additional predicate constants.)

In this paper, a *(Datalog) program* is a pair $(F, \mathbf{p})$, where F is a conjunction of rules, and $\mathbf{p}$ is a tuple of distinct predicate constants.[6] The members of $\mathbf{p}$ are called the *intensional predicates* of the program. The other predicate constants occurring in F are its *extensional predicates*. In many cases, including Examples 1–6, $\mathbf{p}$ is the list of all predicate constants occurring in the heads of the rules of F.

[5] $\widetilde{\forall}F$ stands for the universal closure of F.

[6] In this paper, equality is not considered a predicate constant, so that it is not allowed to be a member of $\mathbf{p}$.

We will define the semantics of Datalog programs by specifying which models of F are considered "stable models" of $(F, \mathbf{p})$. The definition of a stable model is based on a syntactic transformation that turns any Datalog program $(F, \mathbf{p})$ into a second-order sentence, denoted by $\mathrm{SM}_{\mathbf{p}}[F]$. We will define the *stable models* of $(F, \mathbf{p})$ as the models of $\mathrm{SM}_{\mathbf{p}}[F]$ in the sense of second-order logic.[7]

4 Positive Programs

Consider first the simpler case of rules and programs that do not contain intensional predicates in the scope of negation. We will call them *positive*. In Figure 1, the only rules that are not positive are those in lines 4 and 7. In the special case of positive programs, $\mathrm{SM}_{\mathbf{p}}$ is the well-known parallel circumscription operator [12], [2, Section 6.4.2].

The definition of parallel circumscription uses the following notation. If p and q are predicate constants of the same arity then $p \leq q$ stands for the formula $\forall \mathbf{x}(p(\mathbf{x}) \to q(\mathbf{x}))$, where $\mathbf{x}$ is a tuple of distinct object variables. If $\mathbf{p}$ and $\mathbf{q}$ are tuples $p_1, \dots, p_n$ and $q_1, \dots, q_n$ of predicate constants then $\mathbf{p} \leq \mathbf{q}$ stands for the conjunction

$$(p_1 \leq q_1) \wedge \cdots \wedge (p_n \leq q_n).$$

Furthermore, $\mathbf{p} < \mathbf{q}$ stands for $(\mathbf{p} \leq \mathbf{q}) \wedge \neg(\mathbf{q} \leq \mathbf{p})$. This formula expresses that each p_i is a subset of the corresponding q_i, and at least one of these subsets is proper. In second-order logic, we apply the same notation to tuples of predicate variables.

For any positive Datalog program $(F, \mathbf{p})$, we define $\mathrm{SM}_{\mathbf{p}}[F]$ as the sentence

$$F \wedge \neg\exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F(\mathbf{u})), \tag{2}$$

where $\mathbf{u}$ is a list of distinct predicate variables of the same length as $\mathbf{p}$, and $F(\mathbf{u})$ is the formula obtained from F by substituting the variables $\mathbf{u}$ for the constants $\mathbf{p}$.

The second conjunctive term of (2) expresses the minimality of the extents of the predicates $\mathbf{p}$ (with respect to set inclusion) subject to constraint F. Thus the stable models of a positive program $(F, \mathbf{p})$ are the models of F in which $\mathbf{p}$ cannot be made smaller without making F false.

Example 1, continued. Let F be the conjunction of the first-order formulas in lines 1 and 2 of Figure 1. These formulas express that q is a superset of p, and that q is a transitive relation. The formula $\mathrm{SM}_q[F]$ says in addition that q cannot be made smaller without violating property F. Consequently the stable models of the program from Example 1 can be characterized as the interpretations in which q is the transitive closure of p.

Example 2, continued. Let F be the first-order formula in line 3 of Figure 1. It expresses that the union of q and r covers p. The formula $\mathrm{SM}_{qr}[F]$ says

[7] The semantics of second-order formulas is described, for instance, in [8, Section 1.2.3].

in addition that this property will be lost if we change the interpretation by replacing q and r with their subsets. It is clear that this condition is equivalent to the first-order formula

$$\forall x(p(x) \leftrightarrow q(x) \vee r(x)) \wedge \neg\exists x(q(x) \wedge r(x)).$$

The stable models of the program from Example 2 represent arbitrary partitions of p into disjoint subsets q, r.

Remark 1. Consider the result of addings the facts

```
p(a). q(a). r(a).
```

to the program from Example 2. The corresponding first-order formula is

$$\forall x(p(x) \to q(x) \vee r(x)) \wedge (\top \to p(a)) \wedge (\top \to q(a)) \wedge (\top \to r(a)),$$

and the result of applying SM_{qr} to this formula is equivalent to

$$\forall x(p(x) \leftrightarrow q(x) \vee r(x)) \wedge \forall x(q(x) \wedge r(x) \leftrightarrow x = a).$$

In the presence of the additional facts shown above, minimizing q and r does not make these sets disjoint, and it does not make the disjunction exclusive.

Example 4, continued. Let F be the conjunction of the first-order formulas in lines 3 and 5 of Figure 1. These formulas express that the union of q and r covers p, and that a does not belong to q. The formula $\mathrm{SM}_{qr}[F]$ says in addition that the extents of q and r cannot be made smaller without violating property F. This condition is equivalent to

$$\forall x(p(x) \to q(x) \vee r(x)) \wedge \neg q(a) \wedge \neg\exists x(q(x) \wedge r(x)).$$

The stable models of the program from Example 4 represent arbitrary partitions of p into disjoint subsets q, r such that a is not in q.

Example 6, continued. Let F be the conjunction of the first-order formulas in lines 8–10 of Figure 1. These formulas express that r_0 contains all terminal vertices, that r_1 contains all vertices of out-degree 1, and that r_2 contains all vertices of out-degree 2. The result of applying the operator $\mathrm{SM}_{r_0 r_1 r_2}$ to this formula expresses that the sets r_i are minimal subject to these conditions. In the stable models of the program from Example 6, r_0 is the set of terminal vertices, r_1 is the set of vertices of out-degree 1, and r_2 is the set of vertices of out-degree 2.

5 General Definition of a Stable Model

Sentence (2) can be formed even if the Datalog program $(F, \mathbf{p})$ is not positive. But for a nonpositive program the models of that sentence usually match neither the intended meaning of the program nor the behavior of ASP solvers.

This discrepancy can be resolved by modifying (2) as follows. Let $p_1, \dots, p_n$ be the members of the list $\mathbf{p}$, and let $u_1, \dots, u_n$ be the corresponding members of $\mathbf{u}$. By $F^\diamond(\mathbf{u})$ we denote the formula obtained from F by replacing each part $p_i(\mathbf{t})$ that does not belong to the scope of any negation with $u_i(\mathbf{t})$; here $\mathbf{t}$ is an arbitrary tuple of terms. For any Datalog program $(F, \mathbf{p})$, $\mathrm{SM}_{\mathbf{p}}[F]$ stands for the sentence

$$F \wedge \neg\exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^\diamond(\mathbf{u})). \tag{3}$$

It is clear that if $(F, \mathbf{p})$ is positive then $F^\diamond(\mathbf{u})$ is identical to the result $F(\mathbf{u})$ of substituting $\mathbf{u}$ for $\mathbf{p}$ in F. Consequently the new definition of $\mathrm{SM}_{\mathbf{p}}$ is a generalization of the definition from Section 4.

Example 3, continued. Let F be the first-order formula in line 4 of Figure 1. Then $\mathrm{SM}_q[F]$ is

$$\forall x(p(x) \to q(x) \vee \neg q(x)) \wedge \neg\exists u((u < q) \wedge \forall x(p(x) \to u(x) \vee \neg q(x))). \tag{4}$$

Note the disjunction $u(x) \vee \neg q(x)$ at the end of the formula; the occurrence of q in the second disjunctive term is not replaced with u because it is in the scope of a negation. The first conjuctive term of (4) is logically valid, so that it can be dropped. The second term says that the intersection of p and q is not contained in any proper subset of q. This is equivalent to saying that this intersection is itself not a proper subset of q, that is, to the formula $q \leq p$. In the stable models of the program from Example 3, q is an arbitrary subset of p.

Example 5, continued. Let F be the conjunction of the first-order formulas in lines 6 and 7 of Figure 1. Then $\mathrm{SM}_{rs}[F]$ is

$$\begin{array}{l}\forall xy(q(x,y) \to r(x)) \wedge \forall x(p(x) \wedge \neg r(x) \to s(x)) \\ \quad \wedge \neg\exists uv(((u,v) < (r,s)) \wedge \forall xy(q(x,y) \to u(x)) \wedge \forall x(p(x) \wedge \neg r(x) \to v(x))).\end{array} \tag{5}$$

Note that $r(x)$ in the second line did not become $u(x)$: it is in the scope of a negation. Since the subformula $\forall xy(q(x,y) \to u(x))$ does not contain v, and the subformula $\forall x(p(x) \wedge \neg r(x) \to v(x))$ does not contain u, (5) can be rewritten as

$$\begin{array}{l}\forall xy(q(x,y) \to r(x)) \wedge \forall x(p(x) \wedge \neg r(x) \to s(x)) \\ \qquad \wedge \neg\exists u((u < r) \wedge \forall xy(q(x,y) \to u(x))) \\ \qquad \wedge \neg\exists v((v < s) \wedge \forall x(p(x) \wedge \neg r(x) \to v(x))).\end{array}$$

This formula expresses, first, that each nonterminal vertex belongs to r, and that r is the smallest set with this property; second, that s contains the complement of r, and that s is the smallest set with this property. In the stable models of the program from Example 5, r is the set of nonterminal vertices, and s is its complement—the set of terminal vertices.

Remark 2. The definition of a stable model above looks very different from the definition proposed in [5], which involves grounding, constructing the reduct, and checking a fixpoint condition. But it is actually a generalization of the 1988 definition (limited to finite programs); see [3, Corollary 1]. The 1988 definition corresponds to the special case when

– the head of each rule is an atom,
– the body of each rule is a conjunction of literals,
– all predicate constants are intensional,
– we are interested in Herbrand interpretations only.

Remark 3. The definition above differs from the definition of a stable model from [3] in two ways. It is limited to "Datalog programs"—conjuctions of rules; the definition from [3] is applicable to arbitrary first-order sentences. On the other hand, it uses the transformation $F \mapsto F^{\diamond}(\mathbf{u})$ instead of the more complex transformation $F \mapsto F^{*}(\mathbf{u})$ from that paper. (This complexity is the price that one has to pay for the additional generality—for allowing arbitrary first-order sentences as arguments of $\mathrm{SM}_{\mathbf{p}}$.) In application to Datalog programs, the two definitions are equivalent.

6 Equivalent Transformations of Datalog Programs

Recall that the definition of $\mathrm{SM}_{\mathbf{p}}[F]$ for positive F given in Section 4 uses the notation $F(\mathbf{u})$ for the formula obtained from F by substituting the predicate variables $\mathbf{u}$ for the predicate constants $\mathbf{p}$. It is clear that if formulas F_1 and F_2 are equivalent to each other then the formulas $F_1(\mathbf{u})$ and $F_2(\mathbf{u})$ are equivalent to each other as well. It follows that for any positive and equivalent F_1, F_2, the formula $\mathrm{SM}_{\mathbf{p}}[F_1]$ is equivalent to $\mathrm{SM}_{\mathbf{p}}[F_2]$. More generally, if F_1 and F_2 are equivalent to each other and positive then $\mathrm{SM}_{\mathbf{p}}[F_1 \wedge G]$ is equivalent to $\mathrm{SM}_{\mathbf{p}}[F_2 \wedge G]$ for any conjunction G of rules. In other words, replacing a group of positive rules within a Datalog program with an equivalent group of positive rules does not affect the class of stable models of the program.

But without the assumption that the rules involved in the replacement are positive this assertion would be incorrect. For instance, replacing the fact

```
p(a).
```

where `p` is an intensional predicate with the constraint

```
:- not p(a).
```

can change the stable models of the program, even though these rules, written as first-order formulas

$$\top \to p(a),\ \neg p(a) \to \bot, \tag{6}$$

are equivalent to each other. The reason is that the transformation $F \mapsto F^{\diamond}(\mathbf{u})$, applied to two equivalent formulas, may produce non-equivalent formulas. For instance, in application to formulas (6) this transformation gives the non-equivalent formulas

$$\top \to u(a),\ \neg p(a) \to \bot.$$

The results of [15, 9, 10, 3] show, on the other hand, that replacing a group of rules within a Datalog program with another group of rules does not affect the class of stable models whenever the two sets of rules are *intuitionistically* equivalent.[8] (Formulas (6) are equivalent to each other classically, but not intuitionistically.)

We can say even more: Datalog programs $(F_1, \mathbf{p})$ and $(F_2, \mathbf{p})$ have the same stable models if $F_1 \leftrightarrow F_2$ is intuitionistically entailed by the sentences

$$\widetilde{\forall}(F \vee \neg F) \tag{7}$$

for formulas F that do not contain members of the list $\mathbf{p}$.[9]

Compare, for instance, the rule

```
q(X) ; r(X) :- p(X).
```

from Example 2 and the rule

```
q(X) :- p(X), not r(X).
```

The corresponding formulas

$$\forall x(p(x) \rightarrow q(x) \vee r(x)),\ \forall x(p(x) \wedge \neg r(x) \rightarrow q(x)) \tag{8}$$

are not intuitionistically equivalent to each other; it is not surprising then that replacing one rule by the other within a Datalog program usually changes the class of stable models. But the rules above are interchangeable if r is an extensional predicate, because the equivalence between formulas (8) is intuitionistically entailed by

$$\forall x(r(x) \vee \neg r(x)).$$

7 Discussion

The definition of a stable model based on a modification of the circumscription operator provides a declarative semantics for several constructs used in answer set programming, including choice and negation as failure.

Two classes of constructs are conspicuously absent, however, from the examples studied in this paper. One is built-in functions and predicates, such as operations on integers. The other includes aggregates other than `#count`, such as `#sum` (the sum of a set of integers). It appears that such "difficult" aggregates can be handled by extending the operator SM to expressions more general than first-order formulas [4].

Acknowledgements

I am grateful to Paolo Ferraris, Joohyung Lee, Yuliya Lierler, Fangkai Yang, and the anonymous referee for useful comments. This work was partially supported by the National Science Foundation under grant IIS-0712113.

[8] See [13] for an introduction to intuitionistic logic.

[9] This assertion will remain true if we allow F in (7) to have occurrences of intensional predicates as long as each of them is in the scope of a negation or in the antecedent of an implication.

References

1. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. Gerhard Brewka, Ilkka Niemelä, and Mirosław Truszczyński. Nonmonotonic reasoning. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*. Elsevier, 2008.
3. Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.
4. Paolo Ferraris and Vladimir Lifschitz. The stable model semantics for first-order formulas with aggregates[10]. In *Proceedings of International Workshop on Nonmonotonic Reasoning (NMR)*, 2010.
5. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.
6. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
7. Vladimir Lifschitz. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1594–1597. MIT Press, 2008.
8. Vladimir Lifschitz, Leora Morgenstern, and David Plaisted. Knowledge representation and classical logic. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 3–88. Elsevier, 2008.
9. Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
10. Vladimir Lifschitz, David Pearce, and Agustin Valverde. A characterization of strong equivalence for logic programs with variables. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2007.
11. Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.
12. John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986.
13. Joan Moschovakis. Intuitionistic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2008 edition, 2008. `http://plato.stanford.edu/archives/fall2008/entries/logic-intuitionistic`.
14. Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.
15. 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, 1997.

[10] `http://userweb.cs.utexas.edu/users/vl/papers/smaf.pdf`