Order and Negation as Failure

Davy Van Nieuwenborgh* and Dirk Vermeir**

Dept. of Computer Science
Vrije Universiteit Brussel, VUB
{dvnieuwe,dvermeir}@vub.ac.be

Abstract. We equip ordered logic programs with negation as failure, using a simple generalization of the preferred answer set semantics for ordered programs. This extension supports a convenient formulation of certain problems, which is illustrated by means of an intuitive simulation of logic programming with ordered disjunction. The simulation also supports a broader application of "ordered disjunction", handling problems that would be cumbersome to express using ordered disjunction logic programs.
Interestingly, allowing negation as failure in ordered logic programs does not yield any extra computational power: the combination of negation as failure and order can be simulated using order (and true negation) alone.

1 Introduction

Non-monotonic reasoning using logic programming can be accomplished using one of several possible extensions of positive programs. The better known extension is *negation as failure* which has a long history, starting from the Clark completion [7], over stable model semantics [11] and well-founded semantics [25], to answer set programming [20]. It is well-known that adding negation as failure to programs results in a more expressive formalism. However, in the context of disjunctive logic programming [21, 19], [14] demonstrated that adding negation as failure positively in a program, i.e. in the head of the rules, yields no extra computational power to the formalism. One of the more interesting features of negation as failure in the head is that answers no longer have to be minimal w.r.t. subset inclusion (e.g. the program $\{a \vee \mathit{not}\ a \leftarrow\}$ has both $\{a\}$ and $\emptyset$ as answer sets). Indeed, such minimality turns out to be too demanding to express certain problems, e.g. in the areas of abductive logic programming [15, 13] or logic programming with ordered disjunction [1, 4].

Introducing (preference) order in logic programs represents another way to naturally express many "non-monotonic" problems. Many proposals [17, 18, 24, 6, 3, 5, 28, 8, 27, 1] for logic programming extensions incorporate some kind of order, sometimes in a rather subtle way.

The preferred answer set semantics defined in [27], uses a partial order defined among the rules of a simple program, i.e. a non-disjunctive program containing only

* Supported by the FWO.
** This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP project.

classical negation, to prefer certain extended answer sets of the program above others, where the extended answer sets semantics naturally extends the classical one [20] to deal with inconsistencies in a program by allowing contradictory rules to defeat each other. It turns out that such an order can simulate negation as failure in both seminegative logic programs, where only rule bodies may contain negation as failure, under the stable model semantics [11], and disjunctive logic programs under the possible model semantics [23], demonstrating that order is at least as expressive as negation as failure.

Often, the introduction of order increases the expressiveness of a logic programming formalism, as illustrated by the complexity results in e.g. [16, 6, 3, 27]. A natural question to ask then is whether combining order and negation as failure yields even more expressiveness. For the case of the ordered programs from [27], this paper will show that the answer to the above question is negative.

In this paper, we first extend the preferred answer set semantics for ordered programs to extended ordered programs, i.e. programs containing both classical negation and negation as failure combined with an order relation among the rules. Just as for disjunctive logic programming, adding negation as failure positively results in a formalism where answer sets are not anymore guaranteed to be subset minimal. Then we show, by means of a transformation, that the preferred answer set semantics for such extended ordered programs can be captured by the one for ordered programs (without negation as failure).

Despite the fact that extended ordered programs do not yield any extra computational power, they can be used profitably to express certain problems in a more intuitive and natural way. We demonstrate this by providing an elegant transformation from logic programs with ordered disjunction into extended ordered programs.

Logic programming with ordered disjunction [1] is a combination of qualitative choice logic [2] and answer set programming [20]. Instead of an order relation on the rules in a program, this formalism uses an order among the head literals of disjunctive rules. Intuitively, this relation, called ordered disjunction, ranks the conclusions in the head of rules. A lesser alternative should only be chosen if all higher ranked options could not be fulfilled.

Preferred answer sets for programs with ordered disjunction need not be subset minimal. E.g. for the program ($\times$ is used for disjunction and more preferred alternatives come first in the head of a rule)

$$\begin{aligned} a \times b &\leftarrow \\ c \times b &\leftarrow a \\ \neg c &\leftarrow \end{aligned}$$

both $S = \{b, \neg c\}$ and $T = \{a, b, \neg c\}$ are preferred answer sets, although $S \subset T$. Hence, a translation to a class of programs with subset minimal (preferred) answer sets is impossible unless e.g. extra atoms are introduced.

In [4] a procedure to compute the preferred answer sets of a logic program with ordered disjunction is presented. The algorithm uses two different programs: one to generate candidate answer sets and another one to test whether those candidates are preferred w.r.t. the order on the literals in the ordered disjunctions. Using our transformation into extended ordered logic programs, combined with the further translation that removes

negation as failure, the algorithm to compute answer sets for ordered programs [27] can be used to do the same for logic programs with ordered disjunction.

Another advantage of translating ordered disjunction programs to ordered programs is that it becomes possible to conveniently express, using ordered programs, problems that would be cumbersome to do with ordered disjunction. As an example, suppose that, when thirsty, we prefer lemonade upon cola upon coffee upon juice, which is easily expressed using a single rule with ordered disjunction

$$lemonade \times cola \times coffee \times juice \leftarrow thirsty \ .$$

If, however, there are extra conditions attached to some of the choices, things rapidly get more complicated. E.g., suppose that we only like cola if it is warm outside and we only like coffee if we are tired. Representing these conditions needs four rules with ordered disjunctions.

$$\begin{aligned} lemonade \times juice &\leftarrow thirsty, not\ warm, not\ tired \\ lemonade \times cola \times juice &\leftarrow thirsty, warm, not\ tired \\ lemonade \times coffee \times juice &\leftarrow thirsty, not\ warm, tired \\ lemonade \times cola \times coffee \times juice &\leftarrow thirsty, warm, tired \end{aligned}$$

In general, we get an exponential increase in program size; e.g. adding a similar condition on *lemonade* would result in eight rules. Using extended ordered logic programs to simulate ordered disjunction does not suffer such size increases. In fact, an extra condition only has to be added to the body of a single rule in the ordered program.

The paper is organized as follows: Section 2 generalizes simple programs with negation as failure, in both the head and body of rules. Section 3 presents the corresponding extension of ordered programs and their preferred answer set semantics. It is shown that adding negation as failure does not increase the expressiveness of ordered programs, by providing a transformation that eliminates negation as failure from such extended ordered programs. The usefulness of negation as failure in ordered programs is illustrated in Section 4 where an intuitive semantics-preserving transformation from programs with ordered disjunction to extended ordered programs is demonstrated. Conclusions and directions for further research are stated in Section 5. All proofs can be found in [26].

2 Extended Answer Sets for Extended Programs

We use the following basic definitions and notation. A *literal* is an *atom* a or a negated atom $\neg a$. An *extended literal* is a literal or a *naf-literal* of the form *not* l where l is a literal. The latter form denotes negation as failure: *not* l is interpreted as "l is not true". We use $\hat{l}$ to denote the ordinary literal underlying an extended literal, i.e. $\hat{l} = a$ if $l = not\ a$ while $\hat{l} = a$ if $l = a$, a a literal. Both notations are extended to sets so $\hat{X} = \{\hat{e} \mid e \in X\}$, with X a set of extended literals, while $not\ Y = \{not\ l \mid l \in Y\}$ for any set of (ordinary) literals Y.

For a set of literals X we use $\neg X$ to denote $\{\neg p \mid p \in X\}$ where $\neg(\neg a) \equiv a$. Also, X^+ denotes the positive part of X, i.e. $X^+ = \{a \in X \mid a \text{ is an atom}\}$. The *Herbrand*

base of X, denoted $\mathcal{B}_X$, contains all atoms appearing in X, i.e. $\mathcal{B}_X = (X \cup \neg X)^+$. A set I of literals is *consistent* if $I \cap \neg I = \emptyset$. We use X^- to denote the literals underlying elements of X that are not ordinary literals, i.e. $X^- = \{l \mid \mathit{not}\ l \in X\}$. It follows that X is consistent iff the set of ordinary literals $\neg(X^-) \cup (X \setminus \mathit{not}\ X^-)$ is consistent.

Before studying the effects of allowing negation as failure in ordered programs, we first extend the simple logic programs introduced in [27] to support negation as failure in both the head and the body of rules.

Definition 1. *An **extended logic program** (ELP) is a countable set P of **extended rules** of the form $\alpha \leftarrow \beta$ where $\alpha \cup \beta$ is a finite set of extended literals, and $|\alpha| \leq 1$, i.e. α is a singleton or empty. If $(\alpha \cup \beta)^- = \emptyset$, i.e. all rules are free from naf-literals, P is called a **simple program** (SLP).*

We will often confuse a singleton set with its sole element, writing rules as $a \leftarrow \beta$ or $\leftarrow \beta$. The *Herbrand base* $\mathcal{B}_P$ of an ELP P contains all atoms appearing in P. An *interpretation* I of P is any consistent subset of $\mathcal{B}_P \cup \neg\mathcal{B}_P$. An interpretation I is *total* if $\mathcal{B}_P \subseteq I \cup \neg I$.

An extended literal l is true w.r.t. an interpretation I, denoted $I \models l$ if $l \in I$ in case l is ordinary, or $I \not\models a$ if $l = \mathit{not}\ a$ for some ordinary literal a. As usual, $I \models X$, for some set of (extended) literals X, iff $\forall l \in X \cdot I \models l$.

A rule $r = a \leftarrow \beta$ is *satisfied* by I, denoted $I \models r$, if $I \models a$, $a \neq \emptyset$, whenever $I \models \beta$, i.e. if r is *applicable* ($I \models \beta$), then it must be *applied* ($I \models \beta \cup a$).

For a simple program P, an *answer set* is a minimal interpretation I that is *closed* under the rules of P (i.e. $\forall r \in P \cdot I \models r$).

For an ELP P containing negation as failure and an interpretation I, the Gelfond-Lifschitz transformation [11] yields the *GL-reduct* program P^I that consists of those rules $(a \setminus \mathit{not}\ a^-) \leftarrow (\beta \setminus \mathit{not}\ \beta^-)$ where $a \leftarrow \beta$ is in P, $I \models \mathit{not}\ \beta^-$ and $I \models a^-$.

Thus, P^I is obtained from P by (a) removing all true naf-literals $\mathit{not}\ a$, $a \notin I$, from the bodies of rules in P, (b) removing all false naf-literals $\mathit{not}\ a$, $a \in I$ from the heads of rules in P, and (c) keeping in P^I only the transformed rules that are free from negation as failure. An interpretation I is then an *answer set* of P iff I is an answer set of the reduct P^I.

Another reduct w.r.t. an interpretation denotes the set of rules that are satisfied w.r.t. that interpretation.

Definition 2. *The **reduct** $P_I \subseteq P$ of an ELP P w.r.t. an interpretation I contains just the rules satisfied by I, i.e. $P_I = \{r \in P \mid I \models r\}$.*

Naturally, $P_M = P$ for any answer set M of P. Inconsistencies in ELP's are handled by considering a *defeat* relation between rules. In an *extended answer set*, all rules that are not satisfied are defeated.

Definition 3. *An extended rule $r = a \leftarrow \alpha$ is **defeated** w.r.t. an interpretation I of an ELP P iff there exists an applied **competing rule** $r' = a' \leftarrow \alpha'$ such that $\{a, a'\}$ is inconsistent. An interpretation I is an **extended answer set** of P iff I is an answer set of P_I and each unsatisfied rule from $P \setminus P_I$ is defeated w.r.t. I.*

Thus, to verify a candidate extended answer set M for a program P, one first obtains P_M by eliminating unsatisfied rules (verifying that they are defeated) and then checks

that M is a minimal closure of the positive program $(P_M)^M$, obtained by applying the Gelfond-Lifschitz transformation for P_M.

Intuitively, as in [27], Definition 3 handles inconsistencies by allowing one of two contradictory rules (whose heads are inconsistent) to defeat the other.

Example 1. Consider the extended program P containing the following rules.

$$\begin{array}{lll} \neg a \leftarrow & \neg b \leftarrow & c \leftarrow \\ a \leftarrow \mathit{not}\ b & b \leftarrow \mathit{not}\ a & \mathit{not}\ c \leftarrow a \end{array}$$

For the interpretation $I = \{a, \neg b\}$, P_I contains all rules but $\neg a \leftarrow$ and $c \leftarrow$ which are defeated (w.r.t. I) by the applied rules $a \leftarrow \mathit{not}\ b$ and $\mathit{not}\ c \leftarrow a$, respectively. I is then an extended answer set because $\{a, \neg b\}$ is an answer set of $(P_I)^I = \{\neg b \leftarrow,\ a \leftarrow\}$.

P has three more extended answer sets, namely $J = \{\neg a, b, c\}$, $K = \{\neg a, \neg b, c\}$ and $L = \{a, \neg b, c\}$. Here, $P_J = P \backslash \{\neg b \leftarrow\}$, and $(P_J)^J$ contains $\neg a \leftarrow$, $b \leftarrow$, $c \leftarrow$ and $\leftarrow a$. For K, we have that $P_K = P \setminus \{a \leftarrow \mathit{not}\ b,\ b \leftarrow \mathit{not}\ a\}$ and $(P_K)^K$ contains $\neg a \leftarrow$, $\neg b \leftarrow$, $c \leftarrow$ and $\leftarrow a$. Finally, for L, $P_L = P \setminus \{\neg a \leftarrow, \mathit{not}\ c \leftarrow a\}$ and $(P_L)^L$ contains $a \leftarrow$, $\neg b \leftarrow$ and $c \leftarrow$.

Adding negation as failure invalidates some properties holding for simple programs [27].

- Unlike for simple programs, extended answer sets for extended logic programs are not necessary minimal w.r.t. subset inclusion, as demonstrated by the previous example where $I \subset L$. The same holds for extended disjunctive logic programs as shown in [12].
- Every simple program that does not contain constraints (rules with an empty head) has an answer set but an extended program, e.g. $\{a \leftarrow \mathit{not}\ a\}$, may not have any answer set, making the semantics of Definition 3 not universal.
- If a simple program has "traditional" answer sets [20], these coincide with its extended answer sets [27]. For an extended program P, it is clear that any traditional answer set (for which $P_M = P$) is an extended answer set according to Definition 3. However, it may be that, while P has traditional answer sets, it has additional extended answer sets, as illustrated in the following example.

Example 2. Consider the following program P.

$$\begin{array}{ll} \neg b \leftarrow a & b \leftarrow \mathit{not}\ b \\ a \leftarrow \mathit{not}\ b & b \leftarrow \mathit{not}\ a \end{array}$$

Clearly, $I = \{b\}$ is a traditional answer set with P^I containing $\neg b \leftarrow a$ and $b \leftarrow$ which has $\{b\}$ as a minimal answer set. Since $P_I = P$, $\{b\}$ is also an extended answer set.

However, also $J = \{a, \neg b\}$ is an extended answer set because: (1) P_J contains all rules but $b \leftarrow \mathit{not}\ b$; the latter rule is defeated (w.r.t. J) by the applied rule $\neg b \leftarrow a$, and (2) $(P_J)^J$ contains just $a \leftarrow$ and $\neg b \leftarrow a$ and thus J is an answer set of P_J.

Clearly, though, J is not an answer set of P. While J is not very intuitive, such sets will be eliminated by the preference relation introduced in Section 3, see Theorem 2.

Note that the program in the above example does not contain negation as failure in the head of a rule. In fact, negation as failure in the heads of rules can be removed by a construction that is similar to the one used in [14] for reducing DLP's with negation as failure in the head to DLP's without.

Definition 4. *For P an ELP, define $E(P)$ as the ELP, without negation as failure in the head, obtained from P by replacing each rule $a \leftarrow \alpha$ by (for a an ordinary literal, $\mathbf{not}_a$ is a new atom) by $a \leftarrow \alpha$, not $\neg a$, not $\mathbf{not}_a$ when a is an ordinary literal; or by $\mathbf{not}_{\hat{a}} \leftarrow \alpha$, not $\hat{a}$ when a is a naf-literal.*

Note that the above definition captures our intuition about defeat: one can ignore an applicable rule $a \leftarrow \alpha$ if it is defeated by evidence for either $\neg a$ or *not* a, thus making either *not* $\neg a$ or *not* $\mathbf{not}_a$ false and the rule $a \leftarrow \alpha, \mathit{not}\ \neg a, \mathit{not}\ \mathbf{not}_a$ not applicable.

The extended answer sets of P can then be retrieved from the traditional answer sets of $E(P)$.

Theorem 1. *Let P be an ELP. Then, S is an extended answer sets of P iff there is an answer set S' of $E(P)$ such that $S = S' \cap (\mathcal{B}_P \cup \neg\mathcal{B}_P)$.*

For example, let $P = \{a \leftarrow,\ \mathit{not}\ a \leftarrow\}$, which has two extended answer sets $\{a\}$ and $\emptyset$. Then $E(P) = \{a \leftarrow \mathit{not}\ \neg a, \mathit{not}\ \mathbf{not}_a,\ \mathbf{not}_a \leftarrow \mathit{not}\ a\}$ which has two traditional answer sets $\{a\}$ and $\{\mathbf{not}_a\}$ corresponding with $\{a\}$ and $\emptyset$.

3 Extended Ordered Programs

Adding a preference order to extended programs, or, equivalently, allowing negation as failure in ordered programs (OLPs, [27]), yields the class of *extended ordered programs*.

Definition 5. *An **extended ordered logic program** (EOLP) is a pair $\langle R, < \rangle$ where R is an extended program and $<$ is a well-founded strict[1] partial order on the rules in R.[2] If R is free from naf-literals, $\langle R, < \rangle$ is called an **ordered logic program** (OLP).*

Intuitively, $r_1 < r_2$ indicates that r_1 is more preferred than r_2. In the examples we will often represent the order implicitly using the format

$$\begin{array}{c} \cdots \\ \hline R_2 \\ \hline R_1 \\ \hline R_0 \end{array}$$

where each R_i, $i \geq 0$, represents a set of rules, indicating that all rules below a line are more preferred than any of the rules above the line, i.e. $\forall i \geq 0 \cdot \forall r_i \in R_i, r_{i+1} \in R_{i+1} \cdot r_i < r_{i+1}$ or $\forall i \geq 0 \cdot R_i < R_{i+1}$ for short.

[1] A strict partial order $<$ on a set X is a binary relation on X that is antisymmetric, anti-reflexive and transitive. The relation $<$ is well-founded if every nonempty subset of X has a $<$-minimal element.

[2] Strictly speaking, we should allow R to be a multiset or, equivalently, have labeled rules, so that the same rule can appear in several positions in the order. For the sake of simplicity of notation, we will ignore this issue in the present paper: all results also hold for the general multiset case.

Example 3. Consider the following extended ordered logic program describing rules pertaining to a fatal shooting incident.

$$\begin{array}{r}
guilty \leftarrow shoot, dead, not\ self_defense \\
\neg guilty \leftarrow shoot, self_defense \\
self_defense \leftarrow threatened \\
not\ self_defense \leftarrow shoot, unarmed \\
\hline
court_unauthorized \leftarrow normal_court, not\ self_defense \\
not\ guilty \leftarrow court_unauthorized \\
not\ \neg guilty \leftarrow court_unauthorized \\
\hline
unarmed \leftarrow \quad shoot \leftarrow \quad normal_court \leftarrow \\
threatened \leftarrow \quad dead \leftarrow
\end{array}$$

Here the lower (strongest) level provides the facts of the case (someone killed an unarmed person but the victim threatened the killer). The middle level represents the rules governing the handling of such a case in a normal court: if there is no self-defense, it should refer the case to a higher court and no judgment on the guilt status should be proclaimed, i.e. both *not guilty* and *not* $\neg guilty$ should be true. The rules with the lowest preference (in the highest component) describe general criteria to determine qualifications of the case, i.e. guilt and whether a claim of self-defense can be accepted.

Definition 6. *Let $P = \langle R, < \rangle$ be an EOLP. For subsets R_1 and R_2 of R we define $R_1 \sqsubseteq R_2$ iff $\forall r_2 \in R_2 \setminus R_1 \cdot \exists r_1 \in R_1 \setminus R_2 \cdot r_1 < r_2$. We write $R_1 \sqsubset R_2$ just when $R_1 \sqsubseteq R_2$ and $R_1 \neq R_2$. For M_1, M_2 extended answer sets of R, we define $M_1 \sqsubseteq M_2$ iff $R_{M_1} \sqsubseteq R_{M_2}$. As usual, $M_1 \sqsubset M_2$ iff $M_1 \sqsubseteq M_2$ and $M_1 \neq M_2$.*

*An **answer set** for an EOLP P is any extended answer set of R. An answer set for P is called **preferred** if it is minimal w.r.t. $\sqsubseteq$. An answer set is called **proper** if it satisfies all minimal (according to $<$) rules in R.*

Intuitively, a reduct R_1 is preferred over a reduct R_2 if every rule r_2 which is in R_2 but not in R_1 is "countered" by a stronger rule $r_1 < r_2$ from R_1 which is not in R_2.

Note the difference between Definition 6 and other approaches such as [17], which demand that a stronger rule $r_1 \in R_1 \setminus R_2$ countering a weaker rule $r_1 < r_2 \in R_2 \setminus R_1$, must be applied and have a head that contradicts the head of r_2, thus restricting the effect of the order to "competing" rules.

In [27] it is shown that the relation $\sqsubseteq$ on reducts is a partial order.

Example 4. The program from Example 3 has three extended answer sets, all of which contain the set of facts $F = \{unarmed, shoot, dead, threatened, normal\_court\}$ asserted in the most preferred component of the program: $I_1 = F \cup \{self\_defense, \neg guilty\}$, $I_2 = F \cup \{court\_unauthorized\}$ and $I_3 = F \cup \{court\_unauthorized, guilty\}$.

The corresponding reducts are $P_{I_1} = P \setminus \{not\ self\_defense \leftarrow shoot, unarmed\}$, $P_{I_2} = P \setminus \{self\_defense \leftarrow threatened,\ guilty \leftarrow shoot, dead, not\ self\_defense\}$ and $P_{I_3} = P \setminus \{self\_defense \leftarrow threatened,\ not\ guilty \leftarrow court\_unauthorized\}$. Clearly, both $P_{I_2} \sqsubseteq P_{I_3}$, and $P_{I_1} \sqsubseteq P_{I_3}$ since both I_2 and I_1 satisfy all of the middle rules while I_3 defeats *not guilty* $\leftarrow court\_unauthorized$.

Since P_{I_1} and P_{I_2} are incomparable w.r.t. $\sqsubseteq$, it follows that both I_1 and I_2 are preferred answer sets, fitting our intuition that I_3 is unreasonable. The defense would

probably argue for I_1, which clears its client, while the prosecution might assert I_2, referring the case to a higher court.

Example 5. Reconsider the program from Example 1 with the following preference relation, yielding an extended ordered program $\langle P, < \rangle$.

$$\begin{array}{ccc} \neg a \leftarrow & \neg b \leftarrow & \mathit{not}\ c \leftarrow a \\ \hline a \leftarrow \mathit{not}\ b & b \leftarrow \mathit{not}\ a & c \leftarrow \end{array}$$

The reducts of the answer sets of P are $P_I = P \setminus \{c \leftarrow,\ \neg a \leftarrow\}$, $P_J = P \setminus \{\neg b \leftarrow\}$, $P_K = P \setminus \{a \leftarrow \mathit{not}\ b,\ b \leftarrow \mathit{not}\ a\}$ and $P_J = P \setminus \{\neg a \leftarrow, \mathit{not}\ c \leftarrow a\}$, which are ordered by $P_J \sqsubseteq P_I$, $P_J \sqsubseteq P_K$, $P_L \sqsubseteq P_I$ and $P_L \sqsubseteq P_K$, making both $J = \{\neg a, b, c\}$ and $L = \{a, \neg b, c\}$ preferred over both $I = \{a, \neg b\}$ and $K = \{\neg a, \neg b, c\}$.

An interesting interaction between defeat and negation as failure can occur when default (minimally preferred) rules of the form $\mathit{not}\ a \leftarrow$ are used. At first sight, such rules are useless because *not* a is true by default. However, if present, such rules can also be used to defeat others as in the following example.

Example 6.

$$\begin{array}{c} \mathit{not}\ a \leftarrow \\ \hline a \leftarrow \\ \hline \leftarrow a \end{array}$$

This program has the empty set as its single preferred answer set, its reduct containing the rules $\mathit{not}\ a \leftarrow$ and $\leftarrow a$. Without $\mathit{not}\ a \leftarrow$, it would be impossible to defeat $a \leftarrow$, thus violating $\leftarrow a$ and thus the program would not have any answer sets.

Extended programs can be regarded as EOLP's with an empty order relation.

Theorem 2. *The (traditional) answer sets of an ELP P coincide with the proper preferred answer sets of the EOLP $\langle P, \emptyset \rangle$.*

As mentioned in the introduction, negation as failure can be simulated using order alone. However, from Example 1 (where $I \subset L$ are both preferred answer sets), it follows that (proper) preferred answer sets for EOLP's are not necessarily subset-minimal. On the other hand, it has been shown in [27] that (proper) preferred answer sets for ordered programs (without negation as failure) are subset minimal. Hence, simulating an EOLP with an OLP will necessarily involve the introduction of fresh atoms.

One might be tempted to employ a construction similar to the one used e.g. in [17, 27] for simulating negation as failure using a two-level order. This would involve replacing extended literals of the form $\mathit{not}\ a$ by fresh atoms $\mathbf{not}_a$ and adding "default" rules to introduce $\mathbf{not}_a$.

E.g. the extended program $P = \{a \leftarrow \mathit{not}\ b,\ b \leftarrow \mathit{not}\ a\}$ would be simulated by the ordered program $N(P)$

$$\begin{array}{cc} \mathbf{not}_a \leftarrow & \mathbf{not}_b \leftarrow \\ \hline a \leftarrow \mathbf{not}_b & b \leftarrow \mathbf{not}_a \\ \hline \neg a \leftarrow \mathbf{not}_a & \neg b \leftarrow \mathbf{not}_b \\ \neg\mathbf{not}_a \leftarrow a & \neg\mathbf{not}_b \leftarrow b \end{array}$$

where the rules on the lowest level act as constraints, forcing one of the "default rules" in the top level to be defeated in any proper answer set of $N(P)$. The "constraint rules" also serve to indirectly introduce competition between formally unrelated atoms: in the example, we need e.g. a rule to defeat $\mathbf{not}_a \leftarrow$, based on the acceptance of a.

This does not work, however, since it may introduce unwanted answer sets as for the program $\{a \leftarrow \mathit{not}\ a\}$ which would yield the OLP program

$$\begin{array}{r} \mathbf{not}_a \leftarrow \\ \hline a \leftarrow \mathbf{not}_a \\ \hline \neg a \leftarrow \mathbf{not}_a \\ \neg\mathbf{not}_a \leftarrow a \end{array}$$

which has a (proper) preferred answer set $\{\mathbf{not}_a, \neg a\}$ while the original program has no extended answer sets.

The above examples seem to point to contradictory requirements for the corresponding OLP programs: for the first example, rules implying $\mathbf{not}_a$ should be (indirect) competitors for a-rules while for the second example, the $\mathbf{not}_a$-rule should *not* compete with the a-rule, in order not to introduce spurious answer sets.

The solution is to add not only fresh atoms for extended literals of the form *not* a, but also for ordinary literals. Thus each extended literal l will be mapped to an independent new atom $\phi(l)$. A rule $l \leftarrow \alpha$ will then be translated to $\phi(l) \leftarrow \phi(\alpha)$, which does not compete with any other such rule. Defeat between such rules is however supported indirectly by adding extra rules that encode the consequences of applying such a rule: for an original rule of the form $a \leftarrow \alpha$, a an ordinary literal, we ensure that its replacement $\phi(a) \leftarrow \phi(\alpha)$ can, when applied, indirectly defeat $\phi(\neg a)$- and $\phi(\mathit{not}\ a)$-rules by adding both $\neg\phi(\neg a) \leftarrow \phi(\alpha), \phi(a)$ and $\neg\phi(\mathit{not}\ a) \leftarrow \phi(\alpha), \phi(a)$. Similarly, for an original rule of the form $\mathit{not}\ a \leftarrow \alpha$, a an ordinary literal, a rule $\neg\phi(a) \leftarrow \phi(\alpha), \phi(\mathit{not}\ a)$ will be added along with its replacement $\phi(\mathit{not}\ a) \leftarrow \phi(\alpha)$. Consistency is assured by introducing a new most preferred component containing, besides translated constraints $\leftarrow \phi(\alpha)$ for the original ones, rules of the form $\leftarrow \phi(a), \phi(\mathit{not}\ a)$ and $\leftarrow \phi(a), \phi(\neg a)$. In addition, this new component also contains translations $a \leftarrow \phi(a)$ of the new atoms, that correspond to ordinary literals, back to their original versions.

Negation as failure can then be simulated by introducing "default" rules of the form $\phi(\mathit{not}\ a) \leftarrow$ in a new least preferred component.

Spurious answer sets are prevented, as these new default rules introducing $\phi(\mathit{not}\ a)$, which do not have defeat-enabling accompanying rules as described above, cannot be used to defeat transformed rules of the original program, but only to make them applicable. E.g. the program $\{a \leftarrow \mathit{not}\ a\}$ mentioned above would be translated as

$$\begin{array}{rll} \phi(\mathit{not}\ a) \leftarrow & & \phi(\mathit{not}\ \neg a) \leftarrow \\ \hline \phi(a) \leftarrow \phi(\mathit{not}\ a) & & \\ \neg\phi(\mathit{not}\ a) \leftarrow \phi(\mathit{not}\ a), \phi(a) & & \\ \neg\phi(\neg a) \leftarrow \phi(\mathit{not}\ a), \phi(a) & & \\ \hline \leftarrow \phi(a), \phi(\mathit{not}\ a) & & a \leftarrow \phi(a) \\ \leftarrow \phi(a), \phi(\neg a) & & \neg a \leftarrow \phi(\neg a) \\ \leftarrow \phi(\neg a), \phi(\mathit{not}\ \neg a) & & \end{array}$$

which has no proper preferred answer sets.

Formally, for an EOLP $\langle R, < \rangle$, we define a mapping ϕ translating original extended literals by: $\phi(a) = a'$, $\phi(\neg a) = a'_\neg$, $\phi(not\ a) = \mathbf{not}_a$ and $\phi(not\ \neg a) = \mathbf{not}_{\neg a}$; where for each atom $a \in \mathcal{B}_R$, $a', a_\neg{}', \mathbf{not}_a$ and $\mathbf{not}_{\neg a}$ are fresh atoms. We use $\phi(X)$, X a set of extended literals, to denote $\{\phi(x) \mid x \in X\}$.

Definition 7. *Let $P = \langle R, < \rangle$ be an extended ordered logic program. The OLP version of P, denoted $N_s(P)$, is defined by $N_s(P) = \langle R_n \cup R' \cup R_c, R_c < R'_< < R_n \rangle$, where*

- $R_n = \{\phi(not\ a) \leftarrow \mid a \in \mathcal{B}_R \cup \neg\mathcal{B}_R\}$,
- *R' is obtained from R by replacing each rule*
  - *$a \leftarrow \alpha$, where a is a literal, by the rules $\phi(a) \leftarrow \phi(\alpha)$ and $\neg\phi(\neg a) \leftarrow \phi(\alpha), \phi(a)$ and $\neg\phi(not\ a) \leftarrow \phi(\alpha), \phi(a)$;*
  - *not $a \leftarrow \alpha$ by the rules $\phi(not\ a) \leftarrow \phi(\alpha)$ and $\neg\phi(a) \leftarrow \phi(\alpha), \phi(not\ a)$;*
- $R_c = \{\leftarrow \phi(\alpha) \in R\} \cup \{\leftarrow \phi(a), \phi(not\ a);\ \leftarrow \phi(a), \phi(\neg a);\ a \leftarrow \phi(a) \mid a \in \mathcal{B}_R \cup \neg\mathcal{B}_R\}$.

Furthermore, $R'_<$ stands for the original order on R but defined on the corresponding rules in R'.

Note that $N_s(P)$ is free from negation as failure.

Example 7. The OLP $N_s(P)$, corresponding to the EOLP of Example 5 is shown below.

$\mathbf{not}_a \leftarrow$	$\mathbf{not}_b \leftarrow$	$\mathbf{not}_c \leftarrow$
$\mathbf{not}_{\neg a} \leftarrow$	$\mathbf{not}_{\neg b} \leftarrow$	$\mathbf{not}_{\neg c} \leftarrow$
$a'_\neg \leftarrow$	$b'_\neg \leftarrow$	$\mathbf{not}_c \leftarrow a'$
$\neg a' \leftarrow a'_\neg$	$\neg b' \leftarrow b'_\neg$	$\neg c' \leftarrow a', \mathbf{not}_c$
$\neg\mathbf{not}_{\neg a} \leftarrow a'_\neg$	$\neg\mathbf{not}_{\neg b} \leftarrow b'_\neg$	
$a' \leftarrow \mathbf{not}_b$	$b' \leftarrow \mathbf{not}_a$	$c' \leftarrow$
$\neg a'_\neg \leftarrow \mathbf{not}_b, a'$	$\neg b'_\neg \leftarrow \mathbf{not}_a, b'$	$\neg c'_\neg \leftarrow c'$
$\neg\mathbf{not}_a \leftarrow \mathbf{not}_b, a'$	$\neg\mathbf{not}_b \leftarrow \mathbf{not}_a, b'$	$\neg\mathbf{not}_c \leftarrow c'$
$a \leftarrow a'$	$b \leftarrow b'$	$c \leftarrow c'$
$\neg a \leftarrow a'_\neg$	$\neg b \leftarrow b'_\neg$	$\neg c \leftarrow c'_\neg$
$\leftarrow a', \mathbf{not}_a$	$\leftarrow b', \mathbf{not}_b$	$\leftarrow c', \mathbf{not}_c$
$\leftarrow a'_\neg, \mathbf{not}_{\neg a}$	$\leftarrow b'_\neg, \mathbf{not}_{\neg b}$	$\leftarrow c'_\neg, \mathbf{not}_{\neg c}$
$\leftarrow a', a'_\neg$	$\leftarrow b', b'_\neg$	$\leftarrow c', c'_\neg$

The OLP has two proper preferred answer sets $J' = \{\neg a, b, c, a'_\neg, b', c', \neg a', \mathbf{not}_a, \neg\mathbf{not}_b, \neg b'_\neg, \neg\mathbf{not}_c, \neg c'_\neg, \neg\mathbf{not}_{\neg a}, \mathbf{not}_{\neg b}, \mathbf{not}_{\neg c}\}$ and $L' = \{a, \neg b, c, a', b'_\neg, c', \neg b', \neg\mathbf{not}_a, \neg a'_\neg, \mathbf{not}_b, \neg\mathbf{not}_c, \neg c'_\neg, \mathbf{not}_{\neg a}, \neg\mathbf{not}_{\neg b}, \mathbf{not}_{\neg c}\}$, corresponding to the preferred answer sets J and L of P.

In general, we have the following correspondence.

Theorem 3. *Let $P = \langle R, < \rangle$ be an extended ordered logic program. Then, M is a preferred answer set of P iff there exists a proper preferred answer set M' of $N_s(P)$, such that $M = M' \cap (\mathcal{B}_R \cup \neg\mathcal{B}_R)$.*

Since the construction of $N_s(P)$ is polynomial, it follows that the expressiveness of EOLP is the same as for OLP, i.e. the second level of the polynomial hierarchy [27]. Consequently, order can simulate negation as failure, even if the latter is used in combination with order.

4 Ordered disjunction and ordered programs

Logic programming with ordered disjunction (LPOD) [1, 4] adds a new connective, called ordered disjunction, to logic programming. Using this connective, conclusions in the head of a rule are ordered according to preference. Intuitively, one tries to satisfy an applicable rule by using its most preferred, i.e. best ranked, conclusion.

Definition 8. *A **logic program with ordered disjunction** (LPOD) is a set of rules of the form $a_1 \times \ldots \times a_n \leftarrow \beta$, $n \geq 1$, where the a_i's are (ordinary) literals and β is a finite set of extended literals.*

An ordered disjunctive rule $a_1 \times \ldots \times a_n \leftarrow \beta$ can intuitively be read as: if β is true, then accept a_1, if possible; if not, then accept a_2, if possible; ...; if none of $a_1, \ldots, a_{n-1}$ are possible, then a_n must be accepted.

Similarly to ordered programs, the semantics for LPOD's is defined in two steps. First, answer sets for LPOD's are defined, which are then ordered by a relation that takes into account to what degree rules are satisfied. The minimal elements in this ordering are also called preferred answer sets. To avoid confusion, we will use the term "preferred LPOD answer sets" for the latter.

Answer sets for LPOD's are defined using *split programs*, a mechanism first used in [23] to define the possible model semantics for disjunctive logic programs.

Definition 9. *Let $r = a_1 \times \ldots \times a_n \leftarrow \beta$ be a LPOD rule. For $k \leq n$ we define the k^{th} option of r as $r^k = a_k \leftarrow \beta$, not $\{a_1, \ldots, a_{k-1}\}$.*

*An extended logic program P' is called a **split program** of a LPOD P if it is obtained by replacing each rule in P by one of its options. An interpretation S is then an (LPOD) **answer set** of P if it is an answer set of a split program P' of P.*

Note that the split programs defined above do not contain negation as failure in the head of rules.

Example 8. The LPOD

$$\begin{aligned} b \times c \times d &\leftarrow \\ c \times a \times d &\leftarrow \\ \neg c &\leftarrow b \end{aligned}$$

has five answer sets, namely $S_1 = \{a, b, \neg c\}$, $S_2 = \{b, \neg c, d\}$, $S_3 = \{c\}$, $S_4 = \{a, d\}$ and $S_5 = \{d\}$.

Note that $S_5 \subset S_4$, illustrating that answer sets need not be subset-minimal. Intuitively, only S_1 and S_3 are optimal in that they correspond with a best combination of options for each of the rules.

The above intuition is formalized in the following definition of a preference relation on LPOD answer sets.

Definition 10. *Let S be an answer set of a LPOD P. Then S satisfies the rule $a_1 \times \ldots \times a_n \leftarrow \beta$*

- *to degree 1 if $S \not\models \beta$,*
- *to degree j $(1 \leq j \leq n)$ if $S \models \beta$ and $j = min\{i \mid a_i \in S\}$.*

For a set of literals S, we define $S^i(P) = \{r \in P \mid deg_S(r) = i\}$, where $deg_S(r)$ is used to denote the degree to which r is satisfied w.r.t. S.

*Let S_1 and S_2 be answer sets of P. Then S_1 is preferred over S_2, denoted $S_1 \sqsubset S_2$, iff there is a k such that $S_2^k(P) \subset S_1^k(P)$, and for all $j < k$, $S_1^j(P) = S_2^j(P)$. A minimal (according to $\sqsubset$) answer set is called a (LPOD) **preferred answer set** of P.*

Example 9. In the LPOD from Example 8, both S_1 and S_3 satisfy the third rule to degree 1. In addition, S_1 satisfies the first rule to degree 1 and the second rule to degree 2, while S_3 satisfies the second rule to degree 1 and the first rule to degree 2. Thus S_1 and S_3 are incomparable w.r.t. $\sqsubset$. All other answer sets are less preferred than either S_1 or S_3: e.g. S_5 satisfies the first and second rule only to degree 3, and the third rule to degree 1, from which $S_1 \sqsubset S_5$ and $S_3 \sqsubset S_5$. It follows that S_1 and S_3 are both preferred.

The preference relation which is implicit in ordered disjunctive rules can be intuitively simulated using preference between non-disjunctive rules.

Definition 11. *The EOLP version of a LPOD P, denoted $L(P)$, is defined by $L(P) = \langle P_r \cup P_1 \cup \ldots \cup P_n \cup P_d, P_r < P_1 < \ldots < P_n < P_d \rangle$, where*

- *n is the size of the greatest ordered disjunction in P;*
- *P_r contains every non-disjunctive rule $a \leftarrow \beta \in P$, and for every ordered disjunctive rule $a_1 \times \ldots \times a_n \leftarrow \beta \in P$, P_r contains a rule $a_i \leftarrow \beta, not\ \{a_1, \ldots, a_n\} \backslash \{a_i\}$ for every $1 \leq i \leq n$.*
- *P_d contains $not\ a \leftarrow$ for every literal a that appears in the head of some ordered disjunctive rule;*
- *for $1 \leq k \leq n$, P_k is defined by $P_k = \{a_k \leftarrow \beta, not\ \{a_1, \ldots, a_{k-1}\} \mid a_1 \times \ldots \times a_m \leftarrow \beta \in P$ with $k \leq m \leq n\}$.*

Intuitively, the rules in P_r ensure that all rules in P are satisfied, while the rules in P_d allow to defeat a rule in one of the $P_1, \ldots, P_{n-1}$ in favor of a rule in a less preferred component (see also Example 6). Finally, the rules in $P_1, \ldots, P_n$ encode the intuition behind LPOD, i.e. better ranked literals in an ordered disjunction are preferred.

Example 10. The result of the transformation of the LPOD from Example 8 is shown below.

$not\ a \leftarrow$	$not\ b \leftarrow$
$not\ c \leftarrow$	$not\ d \leftarrow$
$d \leftarrow not\ b, not\ c$	$c \leftarrow not\ a, not\ d$
$c \leftarrow not\ b$	$a \leftarrow not\ c$
$b \leftarrow$	$c \leftarrow$
$b \leftarrow not\ c, not\ d$	$c \leftarrow not\ a, not\ d$
$c \leftarrow not\ b, not\ d$	$a \leftarrow not\ c, not\ d$
$d \leftarrow not\ b, not\ c$	$d \leftarrow not\ a, not\ c$
$\neg c \leftarrow b$	

This EOLP program has two proper preferred answer sets, i.e. $S_1 = \{a, b, \neg c\}$ and $S_3 = \{c\}$.

In general, the preferred LPOD answer sets for a LPOD program P coincide with the proper preferred answer sets of $L(P)$.

Theorem 4. *An interpretation S is a preferred LPOD answer set of a LPOD P iff S is a proper preferred answer set of $L(P)$.*

As mentioned in the introduction, LPOD cannot conveniently handle rules of the form $a_1\{\alpha_1\} \times \ldots \times a_n\{\alpha_n\} \leftarrow \beta$, where the $\alpha_1, \ldots, \alpha_n$ are sets of extended literals expressing extra conditions that have to be fulfilled to derive the corresponding conclusion. Intuitively, such a rule means that if β is true, then a_1 should be true if α_1 is also true and if possible; otherwise a_2 should be true if α_2 is also true and if possible; ...; otherwise a_n should be true if α_n is also true and if possible.

The example from Section 1 can then be rewritten as $lemonade \times cola\{warm\} \times coffee\{tired\} \times juice \leftarrow thirsty$. Of course, such kind of rules can be expressed in LPOD, as shown in Section 1, but the resulting programs quickly become unwieldy. On the other hand, expressing the extra condition is easily done for the EOLP version: it suffices to add each α_k to the body of the corresponding rule in P_k, $1 \leq k \leq n$, as shown below.

Example 11. Reconsider the example from Section 1 and suppose that we are thirsty, but there is no lemonade, i.e. consider the program $P = \{lemonade \times cola\{warm\} \times coffee\{tired\} \times juice \leftarrow thirsty; thirsty \leftarrow; \neg lemonade \leftarrow\}$. The EOLP version of this problem is

$$
\begin{array}{rl}
not\ juice \leftarrow & \\
not\ coffee \leftarrow & \\
not\ cola \leftarrow & \\
not\ lemonade \leftarrow & \\
\hline
juice \leftarrow & thirsty, not\ coffee, not\ cola, not\ lemonade \\
\hline
coffee \leftarrow & thirsty, tired, not\ cola, not\ lemonade \\
\hline
cola \leftarrow & thirsty, warm, not\ lemonade \\
\hline
lemonade \leftarrow & thirsty \\
\hline
lemonade \leftarrow & thirsty, not\ cola, not\ coffee, not\ juice \\
cola \leftarrow & thirsty, not\ lemonade, not\ coffee, not\ juice \\
coffee \leftarrow & thirsty, not\ cola, not\ lemonade, not\ juice \\
juice \leftarrow & thirsty, not\ cola, not\ coffee, not\ lemonade \\
thirsty \leftarrow & \\
\neg lemonade \leftarrow &
\end{array}
$$

One can check that this programs has the expected proper preferred answer set $I = \{thirsty, juice, \neg lemonade\}$.

5 Conclusions and Direction for Further Research

Adding negation as failure to ordered programs provides an intuitive formalism to express certain problems. E.g. ordered disjunctive programs can be simulated using such

extended ordered programs, and the simulation is actually more convenient for certain extensions of ordered disjunctive programs. Interestingly, the expressive power of extended ordered programs remains the same as for ordinary ordered programs.

Directions for further research include the practical implementation of the preferred answer set semantics for extended ordered programs, e.g. through a pre-compiler that applies the construction of Definition 7 and feeds the result to an OLP evaluation algorithm as described in [27]. Alternatively, a direct interpretation of EOLP programs may turn out to be more efficient.

In addition, there are some promising application areas for EOLP. E.g. diagnostic problems [22, 9] have a natural representation as extended ordered programs where constraints of the form $o \leftarrow \mathit{not}\ o$ represent observations, slightly less preferred rules model the normal operation and the weakest rules describe the fault model. The proper preferred answer set semantics will then return an explanation for the observations that minimizes recourse to the fault model, which naturally coincides with our intuition of a "best" explanation.

References

1. G. Brewka. Logic programming with ordered disjunction. In *Proc. of the 18th AAAI Conf.*, pages 100–105. AAAI Press, 2002.
2. G. Brewka, S. Benferhat, and D. Le Berre. Qualitative choice logic. In *Proc. of the 8th Intl. Conf. on Knowledge Representation and Reasoning*, pages 158–169. Morgan Kaufmann, 2002.
3. Gerhard Brewka and Thomas Eiter. Preferred answer sets for extended logic programs. *Artifi cial Intelligence*, 109(1-2):297–356, 1999.
4. Gerhard Brewka, Ilkka Niemela, and Tommi Syrjanen. Implementing ordered disjunction using answer set solvers for normal programs. In Flesca et al. [10], pages 444–455.
5. Francesco Buccafurri, Wolfgang Faber, and Nicola Leone. Disjunctive logic programs with inheritance. In Danny De Schreye, editor, *Proc. of the Intl. Conf. on Logic Programming*, pages 79–93. MIT Press, 1999.
6. Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Disjunctive ordered logic: Semantics and expressiveness. In A. et al. Cohn, editor, *Proc. of the 6th Intl. Conf. on Principles of Knowledge Representation and Reasoning*, pages 418–431. Morgan Kaufmann, 1998.
7. Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plemum Press, 1978.
8. Marina De Vos and Dirk Vermeir. Dynamically ordered probabilistic choice logic programming. In Sanjiv Kapoor and Sanjiva Prasad, editors, *Foundations of 20th Software Technology and Theoretical Computer Science, 20th Conference, Proceedings*, volume 1974 of *LNCS*, pages 227–239. Springer Verlag, 2000.
9. Thomas Eiter, Georg Gottlob, and Nicola Leone. Abduction from logic programs: Semantics and complexity. *Theoretical Computer Science*, 189(1-2):129–177, 1997.
10. Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors. *Logic in Artifi cial Intelligence*, volume 2424 of *LNAI*. Springer Verlag, 2002.
11. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In R. et al. Kowalski, editor, *Proc. of the 5th Intl. Conf. on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
12. Katsumi Inoue and Chiaki Sakama. On positive occurrences of negation as failure. In J. at al. Doyle, editor, *Proc. of the 4th Intl. Conf. on Principles of Knowledge Representation and Reasoning*, pages 293–304. Morgan Kaufmann, 1994.

13. Katsumi Inoue and Chiaki Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming*, 27(2):107–136, 1996.
14. Katsumi Inoue and Chiaki Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35(1):39–78, 1998.
15. A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.
16. Robert A. Kowalski and Fariba Sadri. Logic programs with exceptions. In David H. D. Warren and Peter Szeredi, editors, *Proc. of the 7th Intl. Conf. on Logic Programming*, pages 598–613. The MIT Press, 1990.
17. E. Laenens and D. Vermeir. A fixpoint semantics of ordered logic. *Journal of Logic and Computation*, 1(2):159–185, 1990.
18. Els Laenens and Dirk Vermeir. Assumption-free semantics for ordered logic programs: On the relationship between well-founded and stable partial models. *Journal of Logic and Computation*, 2(2):133–172, 1992.
19. Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation*, 135(2):69–112, 1997.
20. Vladimir Lifschitz. Answer set programming and plan generation. *Journal of Artificial Intelligence*, 138(1-2):39–54, 2002.
21. Teodor C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9(3-4):401–424, 1991.
22. Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
23. Chiaki Sakama and Katsumi Inoue. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning*, 13(1):145–172, 1994.
24. Chiaki Sakama and Katsumi Inoue. Representing priorities in logic programs. In Michael J. Maher, editor, *Proc. of the Intl. Conf. on Logic Programming*, pages 82–96. MIT Press, 1996.
25. Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proc. of the 7th PODS Symposium*, pages 221–230. ACM Press, 1988.
26. Davy Van Nieuwenborgh and Dirk Vermeir. Order and negation as failure. Technical report, Vrije Universiteit Brussel, Dept. of Computer Science, 2003.
27. Davy Van Nieuwenborgh and Dirk Vermeir. Preferred answer sets for ordered logic programs. In Flesca et al. [10], pages 432–443.
28. Kewen Wang, Lizhu Zhou, and Fangzhen Lin. Alternating fixpoint theory for logic programs with priority. In John W. Lloyd et al., editor, *Computational Logic*, volume 1861 of *LNCS*, pages 164–178. Springer, 2000.