Logic Programs with Compiled Preferences

James P. Delgrande[1] and **Torsten Schaub**[2] and **Hans Tompits**[3]

Abstract. We describe an approach for compiling dynamic preferences into logic programs under the answer set semantics. An *ordered* logic program is an extended logic program in which rules are named by unique terms, and in which preferences among rules are given by a set of atoms of the form $s \prec t$ where s and t are names. An ordered logic program is transformed into a second, regular, extended logic program wherein the preferences are respected, in that the answer sets obtained in the transformed theory correspond with the preferred answer sets of the original theory. Our approach allows the specification of *static* orderings (in which preferences are external to a logic program), as well as *dynamic* orderings (in which preferences can appear within a program), and orderings on *sets* of rules. In large part then, we are interested in describing a general *methodology* for uniformly incorporating preference information in a logic program. Since the result of our translation is an extended logic program, we can make use of existing implementations, such as `dlv` and `smodels`. To this end, we have developed a compiler, available on the web, as a front-end for these programming systems.

1 INTRODUCTION

In commonsense reasoning one frequently prefers one outcome over another, or the application of one rule over another, or the drawing of one default conclusion over another. For example, in buying a car one may have various desiderata in mind (inexpensive, safe, fast, etc.) where these preferences come in varying degrees of importance. In legal reasoning, laws may apply by default but the laws themselves may conflict. So municipal laws will have a lower priority than state laws, and newer laws will take priority over old. Further, if these preferences conflict, there will be need to invoke higher preferences to decide the conflict.

In this paper we explore the problem of preference orderings within the framework of extended logic programs under the answer set semantics [9]. The general methodology was first proposed in [5], in addressing preferences in default logic. Previous work in dealing with preferences has for the most part treated preference information at the *meta-level* (see Section 6 for a discussion of previous approaches). In contrast, we remain within the framework of extended logic programs: We begin with an *ordered* logic program, which is an extended logic program in which rules are named by unique terms and in which preferences among rules are given by a new set of atoms of the form $s \prec t$, where s and t are names. Thus, preferences among rules are encoded at the *object-level*. An ordered logic program is transformed into a second, regular, extended logic program wherein the preferences are respected, in the sense that the answer sets obtained in the transformed theory correspond to the preferred answer sets of the original theory. The approach is sufficiently general to allow the specification of preferences among preferences, preferences holding in a particular context, and preferences holding by default.

Our approach can be seen as a general *methodology* for uniformly incorporating preference information within a logic program. This transformational approach has several advantages. First, it is flexible. So one can encode how a preference order interacts with other information, or how different types of preference orders (such as specificity, authority, recency, etc.) are to be integrated. Second, it is easier to compare differing approaches handling such orderings, since they can be represented uniformly in the same general setting. Thus, for instance, if someone doesn't like the notion of preference developed here, they may encode their own within this framework. Lastly, it is straightforward implementing our approach: In the present case, we have developed a translator for ordered logic programs that serves as a front-end for the logic programming systems `dlv` [7] and `smodels` [12].

The next section gives background terminology and notation, while Section 3 describes our central approach. Section 4 explores the formal properties of the approach; while Section 5 gives an overview of further features and extensions, and provides a pointer to the implementation. Section 6 compares related work, and Section 7 concludes with a short discussion.

2 DEFINITIONS AND NOTATION

We deal with extended logic programs [11], which allow for expressing both *classical negation* as well as *negation as failure*. We use "$\neg$" for classical negation and "not" for negation as failure. Classical negation is also referred to as *strong negation*, whilst negation as failure is termed *weak negation*.

Our formal treatment is based on propositional languages. As usual, a *literal*, L, is an expression of the form A or $\neg A$, where A is an atom. We assume a possibly infinite set of such atoms. The set of all literals is denoted by Lit. A literal preceded by the negation as failure sign not is said to be a *weakly negated literal*. A *rule*, r, is an expression of the form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n, \qquad (1)$$

where $n \geq m \geq 0$, and each L_i $(0 \leq i \leq n)$ is a literal. The literal L_0 is called the *head* of r, and the set $\{L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n\}$ is the *body* of r. If $n = m$, then r is a *basic rule*; if $n = 0$, then r is a *fact*. An (*extended*) *logic program*, or simply a *program*, is a finite set of rules. A program is *basic* if all rules in

[1] School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada V5A 1S6, email: jim@cs.sfu.ca
[2] Institut für Informatik, Universität Potsdam, Postfach 60 15 53, D–14415 Potsdam, Germany, email: torsten@cs.uni-potsdam.de; also affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada.
[3] Institut für Informationssysteme, Abt. Wissensbasierte Systeme 184/3, Technische Universität Wien, Favoritenstraße 11, A–1040 Wien, Austria, email: tompits@kr.tuwien.ac.at

it are basic. We use $head(r)$ to denote the head of rule r, and $body(r)$ to denote the body of r. Furthermore, let $body^+(r) = \{L_1, \ldots, L_m\}$ and $body^-(r) = \{L_{m+1}, \ldots, L_n\}$. The elements of $body^+(r)$ are referred to as the *prerequisites* of r. We say that a rule r is *defeated* by a set of literals X iff $body^-(r) \cap X \neq \emptyset$. As well, each literal in $body^-(r) \cap X$ is said to *defeat* r.

A set of literals X is *consistent* iff it does not contain a complementary pair A, $\neg A$ of literals. We say that X is *logically closed* iff it is either consistent or equals Lit. Furthermore, X is *closed under* a basic program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body(r) \subseteq X$. The smallest set of literals which is both logically closed and closed under a basic program Π is denoted by $\mathrm{Cn}(\Pi)$.

Let Π be a basic program and X a set of literals. The operator T_Π is defined as follows:

$$T_\Pi X = \{head(r) \mid r \in \Pi, body(r) \subseteq X\}$$

if X is consistent, and $T_\Pi X = Lit$ otherwise. Iterated applications of T_Π are written as T_Π^j ($j \geq 0$), where $T_\Pi^0 X = X$ and $T_\Pi^i X = T_\Pi T_\Pi^{i-1} X$ for $i \geq 1$. It is well-known that $\mathrm{Cn}(\Pi) = \bigcup_{i \geq 0} T_\Pi^i \emptyset$, for any basic program Π.

Let r be a rule. Then r^+ denotes the basic program obtained from r by deleting all weakly negated literals in the body of r, i.e., $r^+ = head(r) \leftarrow body^+(r)$. The *reduct*, Π^X, of a program Π *relative to* a set X of literals is defined by

$$\Pi^X = \{r^+ \mid r \in \Pi \text{ and } r \text{ is not defeated by } X\}.$$

In other words, Π^X is obtained from Π by (i) deleting any $r \in \Pi$ which is defeated by X and (ii) deleting each weakly negated literal occurring in the bodies of the remaining rules. We say that a set X of literals is an *answer set* of a program Π iff $\mathrm{Cn}(\Pi^X) = X$. Clearly, for each answer set X of a program Π, it holds that $X = \bigcup_{i \geq 0} T_{\Pi^X}^i \emptyset$. The answer set semantics for extended logic programs has been defined in [9] as a generalization of the stable model semantics [8] for *general logic programs* (i.e., programs not containing classical negation, $\neg$). The reduct Π^X is often called the *Gelfond-Lifschitz reduction.*

The set Γ_Π^X of all *generating rules* of an answer set X from Π is given by

$$\Gamma_\Pi^X = \{r \in \Pi \mid r^+ \in \Pi^X \text{ and } body^+(r) \subseteq X\}.$$

That is, Γ_Π^X comprises all rules $r \in \Pi$ such that r is not defeated by X and each prerequisite of r is in X. Finally, a sequence $\langle r_i \rangle_{i \in I}$ of rules is *grounded* iff, for all $i \in I$, $\{head(r_j) \mid j < i\}$ is inconsistent, or else $body^+(r_i) \subseteq \{head(r_j) \mid j < i\}$.

3 LOGIC PROGRAMS WITH PREFERENCES

A logic program over a propositional language $\mathcal{L}$ is said to be *ordered* iff $\mathcal{L}$ contains the following pairwise disjoint categories:

- a set $\mathcal{N}$ of terms serving as *names* for rules;
- a set $\mathbf{A}$ of regular (propositional) atoms of a program; and
- a set $\mathbf{A}_\prec$ of *preference atoms* $s \prec t$, where $s, t \in \mathcal{N}$ are names.

For each ordered program Π, we assume furthermore a bijective[4] function $n(\cdot)$ assigning to each rule $r \in \Pi$ a name $n(r) \in \mathcal{N}$. To simplify our notation, we usually write n_r instead of $n(r)$ (and we sometimes abbreviate n_{r_i} by n_i). Also, the relation $t = n(r)$ is written as $t : r$, leaving the naming function $n(\cdot)$ implicit. The elements of $\mathbf{A}_\prec$ express preference relations among rules. Intuitively, $n_r \prec n_{r'}$ asserts that r' has "higher" priority than r. Thus, r' is viewed as having precedence over r. That is, r' should, in some sense, always be considered "before" r.

Most importantly, we impose no restrictions on the occurrences of preference atoms. This allows for expressing preferences in a very flexible, dynamic way. For instance, we may specify

$$n_r \prec n_{r'} \leftarrow p, not\ q$$

where p and q may themselves be (or rely on) preference atoms.

A special case is given by programs containing preference atoms only among their facts. We say that a logic program Π over $\mathcal{L}$ is *statically ordered* if it is of the form $\Pi = \Pi' \cup \Pi''$, where Π' is an ordered logic program over $\mathcal{L} \setminus \mathbf{A}_\prec$ and $\Pi'' \subseteq \{(n_r \prec n_{r'}) \leftarrow \mid r, r' \in \Pi'\}$. The static case can be regarded as being induced from an external order "$<$", where the relation $r < r'$ between two rules holds iff the fact $(n_r \prec n_{r'}) \leftarrow$ is included in the ordered program. We make this explicit by denoting a statically ordered program Π as a pair $(\Pi', <)$, representing the program $\Pi' \cup \{(n_r \prec n_{r'}) \leftarrow \mid r < r'\}$. This static concept of preference corresponds in fact to most previous approaches to preference handling in logic programming and nonmonotonic reasoning, where the preference information is specified as a fixed relation at the meta-level (see, e.g., [1, 2, 13, 4]).

Our approach provides a mapping $\mathcal{T}$ that transforms an ordered logic program Π into a regular logic program $\mathcal{T}(\Pi)$, such that the preferred answer sets of Π are given by the (regular) answer sets of $\mathcal{T}(\Pi)$. Intuitively, the translated program $\mathcal{T}(\Pi)$ is constructed in such a way that the ensuing answer sets respect the inherent preference information induced by the given program Π (see Theorems 3 and 4 below). This is achieved by adding sufficient control elements to the rules of Π which guarantee that successive rule applications are in accord with the intended order.

Given the relation $n_r \prec n_{r'}$, we want to ensure that r' is considered before r, in the sense that, for a given answer set X, rule r' is known to be applied or defeated *ahead of* r (with respect to the grounded enumeration of generating rules of X). We do this by first translating rules so that the order of rule application can be explicitly controlled. For this purpose, we need to be able to detect when a rule has been applied or when a rule is defeated; as well we need to be able to control the application of a rule based on other antecedent conditions. For a rule r, there are two cases for it not to be applied: it may be that some literal in $body^+(r)$ does not appear in the answer set, or it may be that a literal in $body^-(r)$ is in the answer set. For detecting non-applicability (i.e., blockage), we introduce, for each rule r in the given program Π, a new, special-purpose atom $\mathsf{bl}(n_r)$. Similarly, we introduce a special-purpose atom $\mathsf{ap}(n_r)$ to detect the case where a rule has been applied. For controlling application of rule r we introduce the atom $\mathsf{ok}(n_r)$. Informally, we conclude that it is ok to apply a rule just if it is ok with respect to every $\prec$-greater rule; for such a $\prec$-greater rule r', this will be the case just when r' is known to be blocked or applied.

More formally, given an ordered program Π over $\mathcal{L}$, let $\mathcal{L}^+$ be the language obtained from $\mathcal{L}$ by adding, for each $r, r' \in \Pi$, new pairwise distinct propositional atoms $\mathsf{ap}(n_r)$, $\mathsf{bl}(n_r)$, $\mathsf{ok}(n_r)$, and $\mathsf{ok}'(n_r, n_{r'})$. Then, our translation $\mathcal{T}$ maps an ordered program Π over $\mathcal{L}$ into a regular program $\mathcal{T}(\Pi)$ over $\mathcal{L}^+$ in the following way.

Definition 1 *Let $\Pi = \{r_1, \ldots, r_k\}$ be an ordered logic program over $\mathcal{L}$. For each $r \in \Pi$, let $\tau(r)$ be the collection of rules depicted in Figure 1, where $L^+ \in body^+(r)$, $L^- \in body^-(r)$, and $r', r'' \in \Pi$.*

[4] In practice, function n is only required to be injective in order to allow for rules not participating in the resultant preference relation.

Then, the logic program $\mathcal{T}(\Pi)$ over $\mathcal{L}^+$ is given by $\bigcup_{r\in\Pi}\tau(r)$.

The first four rules of Figure 1 express applicability and blocking conditions of the original rules: For each rule $r \in \Pi$, we obtain two rules, $a_1(r)$ and $a_2(r)$, along with n rules of the form $b_1(r, L^+)$ and m rules of the form $b_2(r, L^-)$, where n and m are the numbers of the literals in $body^+(r)$ and $body^-(r)$, respectively. The second group of rules encodes the strategy for handling preferences. The first of these rules, $c_1(r)$, "quantifies" over the rules in Π. This is necessary when dealing with dynamic preferences since preferences may vary depending on the corresponding answer set. The three rules $c_2(r, r')$, $c_3(r, r')$, and $c_4(r, r')$ specify the pairwise dependency of rules in view of the given preference ordering: For any pair of rules r, r' with $n_r \prec n_{r'}$, we derive $\mathsf{ok}'(n_r, n_{r'})$ whenever $n_r \prec n_{r'}$ fails to hold, or whenever either $\mathsf{ap}(n_{r'})$ or $\mathsf{bl}(n_{r'})$ is true. This allows us to derive $\mathsf{ok}(n_r)$, indicating that r may potentially be applied whenever we have for all r' with $n_r \prec n_{r'}$ that r' has been applied or cannot be applied. It is important to note that this is only one of many strategies for dealing with preferences: different strategies are obtainable by changing the specification of $\mathsf{ok}(\cdot)$ and $\mathsf{ok}'(\cdot,\cdot)$.

We have the following characterisation of *preferred answer sets*.

Definition 2 *Let Π be an ordered logic program over language $\mathcal{L}$ and X a set of literals. We say that X is a preferred answer set of Π iff $X = Y \cap \mathcal{L}$ for some answer set Y of $\mathcal{T}(\Pi)$.*

In what follows, answer sets of standard (i.e., unordered) logic programs are also referred to as *regular* answer sets.

As an illustration of our approach, consider the following program Π:

$$\begin{array}{lrcl} r_1 = & \neg a & \leftarrow & \\ r_2 = & b & \leftarrow & \neg a, not\ c \\ r_3 = & c & \leftarrow & not\ b \\ r_4 = & n_3 \prec n_2 & \leftarrow & not\ d \end{array}$$

where n_i denotes the name of rule r_i $(i = 1, \ldots, 4)$. This program has two regular answer sets, one containing b and the other containing c; both contain $\neg a$ and $n_3 \prec n_2$. However, only the first is a preferred answer set. To see this, observe that for any $X \subseteq \{head(r) \mid r \in \mathcal{T}(\Pi)\}$, we have $n_i \prec n_j \notin X$ for each $(i, j) \neq (3, 2)$. We thus get for such X and i, j that $\mathsf{ok}'(n_i, n_j) \in T^1_{\mathcal{T}(\Pi)^X}\emptyset$ by (reduced) rules $c_2(r_i, r_j)^+$, and so $\mathsf{ok}(n_i) \in T^2_{\mathcal{T}(\Pi)^X}\emptyset$ via rule $c_1(r_i)^+ = c_1(r_i)$. Analogously, we get $\mathsf{ap}(n_1), \mathsf{ap}(n_4), \neg a, n_3 \prec n_2$. Now consider the following rules from $\mathcal{T}(\Pi)$:

$$\begin{array}{rcrcl} a_2(r_2) & : & \mathsf{ap}(n_2) & \leftarrow & \mathsf{ok}(n_2), \neg a, not\ c \\ b_1(r_2, \neg a) & : & \mathsf{bl}(n_2) & \leftarrow & \mathsf{ok}(n_2), not\ \neg a \\ b_2(r_2, c) & : & \mathsf{bl}(n_2) & \leftarrow & \mathsf{ok}(n_2), c \\ a_2(r_3) & : & \mathsf{ap}(n_3) & \leftarrow & \mathsf{ok}(n_3), not\ b \\ b_2(r_3, b) & : & \mathsf{bl}(n_3) & \leftarrow & \mathsf{ok}(n_3), b \\ c_3(r_3, r_2) & : & \mathsf{ok}'(n_3, n_2) & \leftarrow & (n_3 \prec n_2), \mathsf{ap}(n_2) \\ c_4(r_3, r_2) & : & \mathsf{ok}'(n_3, n_2) & \leftarrow & (n_3 \prec n_2), \mathsf{bl}(n_2) \end{array}$$

Given $\mathsf{ok}(n_2)$ and $\neg a$, rule $a_2(r_2)$ leaves us with the choice between $c \notin X$ or $c \in X$. First, assume $c \notin X$. We get $\mathsf{ap}(n_2)$ from $a_2(r_2)^+ \in \mathcal{T}(\Pi)^X$. Hence, we get b, $\mathsf{ok}'(n_3, n_2)$, and finally $\mathsf{ok}(n_3)$, which results in $\mathsf{bl}(n_3)$ via $b_2(r_3, b)$. Omitting further details, this yields an answer set containing b while excluding c. Second, assume $c \in X$. This eliminates $a_2(r_2)$ when turning $\mathcal{T}(\Pi)$ into $\mathcal{T}(\Pi)^X$. Also, $b_1(r_2, \neg a)$ is defeated since $\neg a$ is derivable. Rule $b_2(r_2, c)$ is inapplicable, since c is only derivable (from $\mathsf{ap}(n_3)$ via $a_1(r_3)$) in the presence of $\mathsf{ok}(n_3)$. But $\mathsf{ok}(n_3)$ is not derivable since neither $\mathsf{ap}(n_2)$ nor $\mathsf{bl}(n_2)$ is derivable. Since this circular situation is unresolvable, there is no preferred answer set containing c.

$$\begin{array}{rrcl} a_1(r): & head(r) & \leftarrow & \mathsf{ap}(n_r) \\ a_2(r): & \mathsf{ap}(n_r) & \leftarrow & \mathsf{ok}(n_r), body(r) \\ b_1(r, L^+): & \mathsf{bl}(n_r) & \leftarrow & \mathsf{ok}(n_r), not\ L^+ \\ b_2(r, L^-): & \mathsf{bl}(n_r) & \leftarrow & \mathsf{ok}(n_r), L^- \\ c_1(r): & \mathsf{ok}(n_r) & \leftarrow & \mathsf{ok}'(n_r, n_{r_1}), \ldots, \mathsf{ok}'(n_r, n_{r_k}) \\ c_2(r, r'): & \mathsf{ok}'(n_r, n_{r'}) & \leftarrow & not\ (n_r \prec n_{r'}) \\ c_3(r, r'): & \mathsf{ok}'(n_r, n_{r'}) & \leftarrow & (n_r \prec n_{r'}), \mathsf{ap}(n_{r'}) \\ c_4(r, r'): & \mathsf{ok}'(n_r, n_{r'}) & \leftarrow & (n_r \prec n_{r'}), \mathsf{bl}(n_{r'}) \\ t(r, r', r''): & n_r \prec n_{r''} & \leftarrow & n_r \prec n_{r'}, n_{r'} \prec n_{r''} \\ as(r, r'): & \neg(n_{r'} \prec n_r) & \leftarrow & n_r \prec n_{r'} \end{array}$$

Figure 1. Translated rules $\tau(r)$.

4 PROPERTIES OF THE APPROACH

Our first result ensures that the dynamically generated preference information enjoys the usual properties of strict orderings. To this end, we define the following relation: for each set X of literals and every $r, r' \in \Pi$, the relation $r <_X r'$ holds iff $n_r \prec n_{r'} \in X$.

Theorem 1 *Let Π be an ordered logic program and X a consistent answer set of $\mathcal{T}(\Pi)$. Then, $<_X$ is a strict partial order. Moreover, if Π has only static preferences, then $<_X = <_Y$, for any answer set Y of $\mathcal{T}(\Pi)$.*

The following properties shed light on the functioning induced by translation $\mathcal{T}$; they elaborate upon the logic programming operator $T_{\mathcal{T}(\Pi)^X}$ of a reduct $\mathcal{T}(\Pi)^X$:

Theorem 2 *Let X be a consistent answer set of $\mathcal{T}(\Pi)$ for an ordered program Π, and let $\Omega = \mathcal{T}(\Pi)^X$. Then, for any $r \in \Pi$:*

1. *$\mathsf{ok}(n_r) \in X$;*
2. *$\mathsf{ap}(n_r) \in X$ iff $\mathsf{bl}(n_r) \notin X$;*
3. *if r is not defeated by X, $\mathsf{ok}(n_r) \in T^i_\Omega\emptyset$, and $body^+(r) \subseteq T^j_\Omega\emptyset$, then $\mathsf{ap}(n_r) \in T^{\max(i,j)+1}_\Omega\emptyset$;*
4. *$\mathsf{ok}(n_r) \in T^i_\Omega\emptyset$ and $body^+(r) \not\subseteq X$ implies $\mathsf{bl}(n_r) \in T^{i+1}_\Omega\emptyset$;*
5. *if r is defeated by X and $\mathsf{ok}(n_r) \in T^i_\Omega\emptyset$, then $\mathsf{bl}(n_r) \in T^j_\Omega\emptyset$ for some $j > i$;*
6. *$\mathsf{ok}(n_r) \notin T^i_\Omega\emptyset$ implies $\mathsf{ap}(n_r) \notin T^j_\Omega\emptyset$ and $\mathsf{bl}(n_r) \notin T^k_\Omega\emptyset$ for all $j, k < i + 2$.*

The next result shows that the translated rules are considered in accord to the partial order induced by the given preference relation:

Theorem 3 *Let Π be an ordered logic program, X a consistent answer set of $\mathcal{T}(\Pi)$, and $\langle r_i\rangle_{i\in I}$ a grounded enumeration of the set $\Gamma^X_{\mathcal{T}(\Pi)}$ of generating rules of X from $\mathcal{T}(\Pi)$. Then, for all $r, r' \in \Pi$:*

$$\textit{If } r <_X r', \textit{ then } j < i,$$

for all r_i equaling $a_k(r)$ or $b_k(r, L)$, and some r_j equaling $a_{k'}(r')$ or $b_{k'}(r', L')$, with $k, k' = 1, 2$, $L \in body(r)$, and $L' \in body(r')$.

For static preferences, our translation $\mathcal{T}$ amounts to selecting the answer sets of the underlying unordered program that comply with the ordering, $<$.

Definition 3 *Let $(\Pi, <)$ be a statically ordered program. An answer set X of Π is called $<$-preserving if X is either inconsistent, or else there exists a grounded enumeration $\langle r_i\rangle_{i\in I}$ of Γ^X_Π such that, for every $i, j \in I$, we have that:*

1. *if* $r_i < r_j$*, then* $j < i$*; and*
2. *if* $r_i < r'$ *and* $r' \in \Pi \setminus \Gamma_\Pi^X$*, then* $body^+(r') \not\subseteq X$ *or* r' *is defeated by the set* $\{head(r_j) \mid j < i\}$*.*

The next result furnishes semantical underpinnings for statically ordered programs; it provides a correspondence between preferred answer sets and regular answer sets of the original program:

Theorem 4 *Let* $(\Pi, <)$ *be a statically ordered logic program and* X *a set of literals. Then,* X *is a preferred answer set of* $(\Pi, <)$ *iff* X *is a* $<$*-preserving answer set of* Π*.*

This gives rise to the following corollary:

Corollary 1 *Let* $(\Pi, <)$ *and* X *be as in Theorem 4. If* X *is a preferred answer set of* $(\Pi, <)$*, then* X *is an answer set of* Π*.*

Note that the last two results have no counterparts in the general (dynamic) case, due to the lack of a regular answer set of the original program. The preference information is only fully available in the answer sets of the translated program (hence the restriction of the notion of $<$-preservation to the static case).

Also, if no preference information is present, our approach is equivalent to standard answer set semantics. Moreover, the notions of statically ordered and (dynamically) ordered programs coincide in this case.

Theorem 5 *Let* Π *be a logic program over* $\mathcal{L}$ *and* X *a set of literals. If* Π *contains no preference information, i.e. if* $\mathcal{L} \cap \mathbf{A}_\prec = \emptyset$*, then the following statements are equivalent:*

1. X *is a preferred answer set of statically ordered logic program* $(\Pi, \emptyset)$*;*
2. X *is a preferred answer set of ordered logic program* Π*;*
3. X *is a regular answer set of logic program* Π*.*

Recently, Brewka and Eiter [4] suggested two principles, simply termed *Principle I* and *Principle II*, which, they argue, any defeasible rule system handling preferences should satisfy. The next result shows that our approach obeys these principles. However, since the original formulation of Principle I and II is rather generic—motivated by the aim to cover as many different approaches as possible—we must instantiate them in terms of our formalism. It turns out that Principle I is only suitable for statically ordered programs, whilst Principle II admits two guises, one for statically ordered programs, and another one for (dynamically) ordered programs.

Principles I and II, formulated for our approach, are as follows:

Principle I. Let $(\Pi, <)$ be a statically ordered logic program, and let X_1 and X_2 be two (regular) answer sets of Π generated by $R \cup \{r_1\}$ and $R \cup \{r_2\}$, respectively, where $r_1, r_2 \notin R$. If $r_1 < r_2$, then X_1 is not a preferred answer set of $(\Pi, <)$.

Principle II-S (Static Case). Let X be a preferred answer set of statically ordered logic program $(\Pi, <)$, let r be a rule wherein $body^+(r) \not\subseteq X$, and let $<'$ be a strict partial order which agrees with $<$ on rules from Π. Then, $X \cup A$ is an answer set of $(\Pi \cup \{r\}, <')$, where $A = \{(n_r \prec n_s) \mid r <' s\} \cup \{\neg(n_s \prec n_r) \mid r <' s\}$.[5]

Principle II-D (Dynamic Case). Let X be a preferred answer set of a (dynamically) ordered logic program Π, and let r be a rule such that $body^+(r) \not\subseteq X$. Then, X is an answer set of $\Pi \cup \{r\}$.

[5] The inclusion of A is necessary because we encode the preference information at the object level.

Theorem 6 *Statically ordered logic programs obey Principles I and II-S. Furthermore, ordered logic programs enjoy Principle II-D.*

Observe that, since transformation $\mathcal{T}$ is clearly polynomial in the size of ordered logic programs, and because of Theorem 5, the complexity of our approach is inherited from the complexity of standard answer set semantics in a straightforward way. We just note the following result:

Theorem 7 *Given an ordered program* Π*, checking whether* Π *has a preferred answer set is NP-complete.*

5 FURTHER ISSUES AND REFINEMENTS

In this section, we sketch the range of applicability and point out distinguishing features of our approach. We briefly mention two points concerning expressiveness, and then sketch how we can deal with preferences over sets of rules. Lastly, we refer to the implementation of our approach.

First, we draw the reader's attention to the expressive power offered by dynamic preferences in connection with variables in the input language, such as

$$n_1(x) \prec n_2(y) \leftarrow p(y), not\ (x = c), \tag{2}$$

where $n_1(x), n_2(y)$ are names of rules containing the variables x and y, respectively. Although such a rule represents only its set of ground instances, it is actually a much more concise specification. Also, since most other approaches employ static preferences of the form $n_1(x) \prec n_2(y) \leftarrow$, such approaches would necessarily have to express (2) as an enumeration of static ground preferences rather than a single rule.

Second, we note that transformation $\mathcal{T}$ is also applicable to disjunctive logic programs (where rule heads are disjunctions of literals). To see this, observe that the transformed rules unfold the conditions expressed in the body of the rules, while the rules' head remain untouched, as manifested by rule $a_1(r)$.

Third, we have extended the approach to allow for preferences between sets of rules. Although we do not include a full discussion here, we remark that this extension has also been implemented (see below). In order to refer to sets of rules, the language is adjoined by a set $\mathcal{M}$ of terms serving as names for sets of rules, and, in addition, the set $\mathbf{A}_\prec$ may now include atoms of the form $m \prec m'$ with $m, m' \in \mathcal{M}$. Accordingly, *set-ordered programs* contain preference information between names of sets. Informally, set M of rules is applicable iff all its members are applicable. Consequently, if M' is preferred over M, then M is considered after *all* rules in M' are found to be applicable, or some rule in M' is found to be inapplicable. As before, set-ordered programs are translated into standard logic programs, where suitable control elements $\mathsf{ok}(\cdot)$, $\mathsf{bl}(\cdot)$, and $\mathsf{ap}(\cdot)$, ranging over names of sets, take care of the intended ordering information.

As an example, consider where in buying a car one ranks the price (e) over safety features (s) over power (p), but safety features together with power is ranked over price. Taking $r_x = x \leftarrow not\ \neg x$ for $x \in \{e, s, p\}$, we can write this (informally) as:

$$m_1 : \{r_p\} \ < \ m_2 : \{r_s\} \ < \ m_3 : \{r_e\} \ < \ m_4 : \{r_p, r_s\}$$

The terms m_1, m_2, m_3, and m_4 are names of sets of rules. If we were given only that not all desiderata can be satisfied then we could apply the rules in the set (named) m_4 and conclude that p and s can be met. Furthermore, sets of rules are described extensionally by means of atoms $\mathsf{in}(\cdot, \cdot)$. Thus, the set $m_4 : \{r_p, r_s\}$

is captured by $\mathsf{in}(n_p, m_4) \leftarrow$ and $\mathsf{in}(n_s, m_4) \leftarrow$. Accordingly, we have $\mathsf{in}(n_p, m_1) \leftarrow$, $\mathsf{in}(n_s, m_2) \leftarrow$, and $\mathsf{in}(n_e, m_3) \leftarrow$. Given rules r_e, r_p, r_s and the previous facts about in, the specification of our example is completed by the preferences $m_i \prec m_{i+1} \leftarrow$ for $i = 1, 2, 3$.

Besides the discussed extensions, our overall framework is general enough to express other strategies for preference handling, like that proposed in [4]. This instance of our framework is described in a companion paper.

Lastly, the approach has been implemented in Prolog and serves as a front-end to the logic programming systems `dlv` [7] and `smodels` [12]. The current prototype is available at

`http://www.cs.uni-potsdam.de/~torsten/plp/`.

This URL contains also diverse examples taken from the literature. Both the dynamic approach to (single) preferences and the set-based approach have been implemented. We note also that the implementation differs from the approach described here in two respects: first, the translation applies to named rules only, i.e., it leaves unnamed rules unaffected; and second, it provides a module which admits the specification of rules containing variables, whereby rules of this form are processed by applying an additional grounding step. More details on the implemented front-end can be found in [6].

6 RELATED WORK

Dealing with preferences on rules seems to necessitate a two-level approach. This in fact is a characteristic of many approaches found in the literature. The majority of these approaches treat preference at the meta-level by defining alternative semantics. [3] proposes a modification of well-founded semantics in which dynamic preferences may be given for rules employing not. [13] and [4] propose different prioritized versions of answer set semantics. In [13] static preferences are addressed first, by defining the *reduct* of a logic program Π, which is a subset of Π that is most preferred. For the following example, their approach gives two answer sets (one with p and one with $\neg p$) which seems to be counter-intuitive; ours in contrast has a single answer set containing $\neg p$.

$$\begin{array}{lcl} r_1 & = & p \leftarrow not\ q_1 \\ r_2 & = & \neg p \leftarrow not\ q_2 \\ & & r_1 < r_2 \leftarrow \end{array}$$

Moreover, the dynamic case is addressed by specifying a transformation of a dynamic program to a set of static programs.

Brewka and Eiter [4] address static preferences on rules in extended logic programs. They begin with a strict partial order on a set of rules, but define preference with respect to total orders that conform to the original partial order. Preferred answer sets are then selected from among the collection of answer sets of the (unprioritised) program. In contrast, we deal only with the original partial order, which is translated into the object theory. As well, only preferred extensions are produced in our approach; there is no need for meta-level filtering of extensions.

Gelfond and Son [10] propose a special-purpose language for directly encoding preferences in a logic programming setting. To this end, they pursue a "two-level" approach in reifying rules and preferences. For example, a rule like $p \leftarrow r, \neg s, not\ q$ is expressed by the formula $default(n, p, [r, \neg s], [q])$ (or, after reification, by the corresponding *term* inside a *holds*-predicate, respectively) where n is the name of the rule. The semantics of a domain description is given in terms of a set of domain-independent rules for predicates like *holds*. These rules can be regarded as a meta-interpreter for the domain description.

7 CONCLUSION

We have described an approach for compiling preferences into logic programs under the answer set semantics. An ordered logic program, in which preferences appear in the program rules, is transformed into a second, extended logic program wherein the preferences are respected, in that the answer sets obtained in the transformed theory correspond with the preferred answer sets of the original theory. In a certain sense, our transformation can be regarded as an axiomatisation of (our interpretation of) preference. Arguably then, we describe a general *methodology* for uniformly incorporating preference information in a logic program. In this approach, we avoid the two-level structure of previous work. While the previous "meta-level" approaches must commit themselves to a semantics and a fixed strategy, our approach (as well as that of [10]) is very flexible with respect to changing strategies, and is open for adaptation to different semantics and different concepts of preference handling.

The approach is easily restricted to reflect a *static* ordering in which preferences are external to a logic program. We also indicated how the approach can be extended to deal with preferences among sets of rules. Finally, this paper demonstrates that our approach is easily implementable; indeed, we have developed a compiler, as a front-end for `dlv` and `smodels`.

ACKNOWLEDGEMENTS

The second author was partially supported by the German Science Foundation (DFG) under grant FOR 375/1-1, TP C. The third author was partially supported by the Austrian Science Fund (FWF) under grants N Z29-INF and P13871-INF.

REFERENCES

[1] F. Baader and B. Hollunder, 'How to prefer more specific defaults in terminological default logic', in *Proc. IJCAI*, pp. 669–674, (1993).
[2] G. Brewka, 'Adding priorities and specificity to default logic', in *Proc. JELIA*, eds., L. Pereira and D. Pearce, pp. 247–260. Springer, (1994).
[3] G. Brewka, 'Well-founded semantics for extended logic programs with dynamic preferences', *J. Artificial Intelligence Research*, **4**, 19–36, (1996).
[4] G. Brewka and T. Eiter, 'Preferred answer sets for extended logic programs', *Artificial Intelligence*, **109**(1-2), 297–356, (1999).
[5] J. Delgrande and T. Schaub, 'Compiling reasoning with and about preferences into default logic', in *Proc. IJCAI*, ed., M. Pollack, pp. 168–174. Morgan Kaufmann, (1997).
[6] J. Delgrande, T. Schaub, and H. Tompits, 'A compiler for ordered logic programs', in *Proc. NMR*, ed., C. Baral and M. Truszczyński, (2000).
[7] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello, 'A deductive system for nonmonotonic reasoning', in *Proc. LPNMR*, eds., J. Dix, U. Furbach, and A. Nerode, pp. 363–374. Springer, (1997).
[8] M. Gelfond and V. Lifschitz, 'The stable model semantics for logic programming', in *Proc. ICLP*, (1988).
[9] M. Gelfond and V. Lifschitz, 'Classical negation in logic programs and deductive databases', *New Generation Computing*, (1991).
[10] M. Gelfond and T. Son, 'Reasoning with prioritized defaults.', in *Proc. LPKR*, eds., J. Dix, L. Pereira, and T. Przymusinski, pp. 164–223. Springer, (1997).
[11] V. Lifschitz, 'Foundations of logic programming', in *Principles of Knowledge Representation*, ed., G. Brewka, 69–127, CSLI, (1996).
[12] I. Niemelä and P. Simons, 'Smodels: An implementation of the stable model and well-founded semantics for normal logic programs', in *Proc. LPNMR*, eds., J. Dix, U. Furbach, and A. Nerode, pp. 420–429. Springer, (1997).
[13] Y. Zhang and N. Foo, 'Answer sets for prioritized logic programs', in *Proc. ILPS*, ed., J. Maluszynski, pp. 69–84. MIT Press, (1997).