A Compiler for Ordered Logic Programs[*]

James P. Delgrande
School of Computing Science,
Simon Fraser University,
Burnaby, B.C.,
Canada V5A 1S6,
jim@cs.sfu.ca,

Torsten Schaub
Institut für Informatik,
Universität Potsdam,
Postfach 60 15 53,
D–14415 Potsdam, Germany,
torsten@cs.uni-potsdam.de

Hans Tompits
Abt. Wissensbasierte Systeme,
Technische Universität Wien,
Favoritenstraße 9–11,
A–1040 Wien, Austria,
tompits@kr.tuwien.ac.at

Abstract

This paper describes a system, called PLP, for compiling ordered logic programs into standard logic programs under the answer set semantics. In an ordered logic program, preference information is expressed at the object level by atoms of the form $s \prec t$, where s and t are names of rules. 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. Since the result of the translation is an extended logic program, existing logic programming systems can be used as underlying reasoning engine. In particular, PLP is conceived as a front-end to the logic programming systems dlv and smodels.

General Information

Several approaches have been introduced in recent years undertaking the ability to express preference information within declarative knowledge representation formalisms (Baader & Hollunder 1993; Brewka 1994; Brewka 1996; Gelfond & Son 1997; Zhang & Foo 1997; Brewka & Eiter 1999). However, most of these methods treat preferences at the meta-level and require a change of the underlying semantics. For instance, incorporating explicit orderings of logic programming rules is realized by modifying the fixed-point conditions characterizing answer sets or well-founded consequences. As a result, implementations need in general fresh algorithms and cannot rely on existing systems computing the regular (unordered) formalisms.

In this paper, we describe a compiler, PLP, for preferred answer sets which evades the need of new algorithms. PLP is based on an approach for expressing preference information *within* the framework of standard answer set semantics. The general technique is described in (Delgrande, Schaub, & Tompits 2000) and derives from a methodology for addressing preferences in default logic first proposed in (Delgrande & Schaub 1997). 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. Such an ordered logic program is then 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.

PLP has been implemented in Prolog and serves as a front-end for the logic programming systems dlv (Eiter *et al.* 1997) and smodels (Niemelä & Simons 1997); it has been developed under the ECLiPSe Constraint Logic Programming System and comprises roughly 300 lines of code. It runs under any operating systems for which dlv or smodels are executable, provided ECLiPSe Prolog is available. Apart from the pretty-printer, we only employ standard Prolog programming constructs.

Description of the System

The logic programming systems dlv and smodels represent state-of-the-art implementations of the stable model semantics (Gelfond & Lifschitz 1988), the former system also admitting strong negation and disjunctive rules. As well, several front-ends for these systems have been developed with the purpose of providing direct encodings of advanced reasoning problems, like, e.g., diagnostic reasoning or planning (Eiter *et al.* 1999; Eiter *et al.* 2000).

PLP realizes a further front-end for these systems, handling ordered logic programs. The structure of PLP is divided into five files:

- The actual compiler is contained in the file pref.pl (or pref$x$.pl, where x is the actual version number).

- The file grounder.pl takes care of programs with variables and non-atomic term structures.

- The file pp.pl provides a pretty-printer for the files resulting from the compilation. (This module relies on features of ECLiPSe for transforming Prolog's variables like _4711 into more readable ones like X.

This is necessary in view of processing the files with `dlv` and `smodels`.)

- The files `dlv.pl` and `smodels.pl` contain code specific to the respective logic programming systems.

The general function of PLP works in three stages:

1. loading the compiler (which charges also the Prolog files `pp.pl`, `grounder.pl`, `dlv.pl`, and `smodels.pl`);

2. compiling the original Prolog file (the compiler is pretty verbose and displays also intermediate versions of the compiled program);

3. call of `dlv` or `smodels`.

The present version of PLP is written for the following releases of `dlv` and `smodels`:

`dlv` : release from November 24, 1999.

`smodels` : version 1.12, including `parse-1.3`. Adaptions for `lparse` and `pparse` (and thus `smodels-2`) are under development.

The current prototype of PLP can be found at the following URL:

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

Applying the System

Methodology

As part of the general logic programming paradigm, ordered logic programs deal with knowledge representation tasks in a *declarative* way. As well, the methodology of representing problems using ordered logic programs is essentially the same as the methodology of representing problems in terms of standard logic programs. However, the admission of explicit preference relations within an ordered logic programs offers a convenient way to represent problems which would otherwise be somewhat tedious to handle.

Specifics

PLP offers several methods how preferences can be encoded. To begin with, preferences between single *rules* can be specified. As well, programs can contain preference relations between *sets* of (names of) rules. Finally, in addition to the general approach discussed in (Delgrande, Schaub, & Tompits 2000), our compiler can also handle rules with variables.

Preferences Between Single Rules. The specification of preference relations between single rules is achieved by associating names to certain (or all) rules of the given program, and by using atoms of the form $n \prec n'$, where n, n' are names of rules, to express a preference relation between rules named n and n', respectively. Informally, atom $n \prec n'$ states that the rule named n has higher priority than the rule named n'. Preference atoms may occur in any program rules, thus

defining preference information in a *dynamic way*. For instance, the rule

$$n \prec n' \leftarrow p, not\, q$$

expresses preference of rule named n' over rule named n in case that p is known and q is not known. Observe that p and q may themselves be preference atoms, or rely on other preference atoms.

Let us illustrate the use of preference information between single rules by the following program Π, representing the well-known penguin-birds-example, but where an additional preference between two conflicting rules is specified:

$$
\begin{aligned}
penguin(tweety) &\leftarrow \\
bird(tweety) &\leftarrow \\
r_1: \quad flies(tweety) &\leftarrow bird(tweety), \\
&\quad\; not\, \neg flies(tweety) \\
r_2: \quad \neg flies(tweety) &\leftarrow penguin(tweety), \\
&\quad\; not\, flies(tweety) \\
n_1 \prec n_2 &\leftarrow
\end{aligned}
$$

Rules r_1 and r_2 are associated with names n_1 and n_2, which are constants in the object language. It is not necessary to name each rule in a program, but only those which occur as an argument in a preference atom. Fact $n_1 \prec n_2 \leftarrow$ expresses that r_2 has higher priority than r_1. Without this preference information, this program has two answer sets, one containing $flies(tweety)$ and the other containing $\neg flies(tweety)$.

The above program Π is expressed in our syntax as follows:

```
penguin(tweety).
bird(tweety).
flies(tweety) :-  name(1),
                  not neg flies(tweety),
                  bird(tweety).
neg flies(tweety) :-  name(2),
                      not flies(tweety),
                      penguin(tweety).
1 < 2.
```

The atoms `name(1)` and `name(2)` are used to refer to the names of r_1 and r_2, respectively. Compiling the file yields the following result:

```
penguin(tweety).
bird(tweety).
flies(tweety) :- ap(1).
ap(1) :-  ok(1), not not_flies(tweety),
          bird(tweety).
ap(2) :-  ok(2), not flies(tweety),
          penguin(tweety).
ok(N) :-  name(N), oko(N, 1), oko(N, 2).
bl(1) :-  ok(1), not_flies(tweety).
bl(1) :-  ok(1), not bird(tweety).
bl(2) :-  ok(2), flies(tweety).
bl(2) :-  ok(2), not penguin(tweety).
```

```
not_flies(tweety) :- ap(2).
prec(1, 2).
oko(N, N) :- name(N).
oko(N, M) :- name(N), name(M),
             not prec(N, M).
oko(N, M) :- name(N), name(M),
             prec(N, M), ap(M).
oko(N, M) :- name(N), name(M),
             prec(N, M), bl(M).
name(2).
name(1).
```

What is the meaning of the newly introduced predicates and rules? The predicates `ap/1`, `bl/1`, `ok/1`, and `oko/2` control the order of the applications of the original rules and guarantee that the intended preference relation is respected. Informally, `ap(n)` and `bl(n)` express that the rule named `n` is already known to be "applied" or "blocked", respectively, whilst `ok(n)` states that it is acceptable to apply the rule associated with name `n`. Atom `ok(n)` is accepted on the basis of the auxiliary predicates `oko/2`. As well, the order relation `<` is captured by the predicate `prec/2`.

Piping the result into `dlv` yields the following answer set:

```
{true, name(1), name(2), penguin(tweety),
 bird(tweety), ok(2), oko(1,1), oko(2,1),
 oko(2,2), prec(1,2), neg_prec(2,1),
 ap(2), neg_flies(tweety), oko(1,2),
 ok(1), bl(1)}
```

We obtain a single answer set that corresponds to the second answer set for the example without preferences. Also, it includes the additional atoms that stem from the necessary preference-handling rules, as discussed above. If the user desires, these special-purpose atoms may be suppressed from the output by using the option "`nice`" in the execution command.

Dynamic Preferences with Variables. PLP can also deal with parametrized names. The use of variables admits the specification of conditional, context-sensitive preferences. Consider the following example:

```
bird(tweety).  penguin(tweety).  water_shy(tweety).
bird(opus).  emu(opus).
bird(scully).  toy(scully).
    flies(X) :- name(r1(X)), not neg flies(X),
                bird(X).
neg flies(X) :- name(r2(X)), not flies(X),
                penguin(X).
neg flies(X) :- name(r3(X)), not flies(X), emu(X).
neg flies(X) :- name(r4(X)), not flies(X), toy(X).
(r1(X) < r2(X)) :- not water_shy(X).
(r1(X) < r3(X)).
```

For treating such programs, they are preprocessed in two steps:

1. rules with variables are replaced by their ground instances;

2. names with a complex term structure are turned into constants (eg. `name(r(f(c)))` is turned into `name(r_f_c)`).

The above program yields the following answer sets:

```
{bird(tweety), bird(opus), bird(scully),
 penguin(tweety), water_shy(tweety),
 emu(opus), toy(scully), flies(tweety),
 flies(scully), neg_flies(opus)}
```
```
{bird(tweety), bird(opus), bird(scully),
 penguin(tweety), water_shy(tweety),
 emu(opus), toy(scully), flies(scully),
 neg_flies(tweety), neg_flies(opus)}
```
```
{bird(tweety), bird(opus), bird(scully),
 penguin(tweety), water_shy(tweety),
 emu(opus), toy(scully), flies(tweety),
 neg_flies(opus), neg_flies(scully)}
```
```
{bird(tweety), bird(opus), bird(scully),
 penguin(tweety), water_shy(tweety),
 emu(opus), toy(scully), neg_flies(tweety),
 neg_flies(opus), neg_flies(scully)}
```

By using the option "`nice`", we suppressed the special-purpose atoms in the displayed output.

Set-ordered Programs. Generalizing the idea of preferences between rules, *set-ordered programs* admit the specification of preference information between sets of rules. Informally, if set M' is preferred over set 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.

In order to express preferences between sets of rules, set-ordered programs associate unique names to sets of rules and express preference information in terms of atoms $m \prec m'$, where m, m' are names of sets of rules. Following our previous convention, $m \prec m'$ states that the set named m' is preferred over the set named m.

The idea behind sets of preferences is illustrated by the following example dealing with "car features":

> Consider where in buying a car one ranks the price ("`expensive`") over safety features ("`safe`") over power ("`powerful`"), but safety features together with power is ranked over price.

This is expressed by the following set-ordered program:

```
expensive :- name(e), not neg expensive.
powerful  :- name(p), not neg powerful.
safe      :- name(s), not neg safe.

neg expensive :-              powerful, safe.
neg powerful  :- expensive,             safe.
neg safe      :- expensive, powerful      .

m1 :  [p].
m2 :  [s].
m3 :  [e].
m4 :  [p,s].

m1 < m2.
m2 < m3.
m3 < m4.
```

Here, `m1` is the name of the set `[p]` comprising the rule named `p` (and analogously for `m2`, `m3`, and `m4`). More generally, sets are expressed in an extensional way by:

```
<set-name> : [<rule-name>,...,<rule-name>] .
```

As well, preferences on individual rules are expressed via the corresponding singleton sets.

Engaging our compiler and piping the described program into `dlv`, we obtain the following answer set:

```
{powerful, safe, neg_expensive}
```

Again, in the displayed result we suppressed the special-purpose atoms by engaging the option "`nice`".

Users and Usability

As for any logic programming system, proper usage of PLP requires the user's ability to formalize a given problem adequately in terms of an ordered (or set-ordered) logic program. However, as already pointed out above, the added benefit of ordered logic programs to allow the explicit specification of certain domain knowledge in terms of preference declarations enriches the already versatile logic programming paradigm.

Evaluating the System

Benchmarks

Accepted benchmarks for nonmonotonic reasoning systems have been realized by the well-known TheoryBase system (Cholewiński *et al.* 1995). This test-bed provides encodings of various graph problems in terms of default theories and equivalent logic programs. Although these benchmark problems do not include ordered logic programs, they can nevertheless be used to evaluate PLP. The reason for this is given by the fact that our method reduces ordered logic programs to regular logic programs. Moreover, since ordered logic programs agree semantically with regular logic programs if no preference information is present, worst-case classes for regular logic programs are also worst-case classes for ordered logic programs.

Comparison

Comparisons between different theorem provers are often difficult to realize because of incompatibilities of the respective underlying semantics. Even if two systems are based on the same formalism, they may represent different syntactical fragments rendering significant comparisons nigh impossible.

A method to address some of these shortcomings is to perform comparisons taking the representational power of the implemented formalisms into account. That is to say, one chooses some "natural" problems, encodes it with respect to the specific methodologies associated with the implemented formalisms, and uses the resultant instances as queries of the respective systems. So, basically, different systems are compared on the basis of (possibly) different representations *of the same problem.* In this sense, PLP can be compared with other logic programming systems, whether or not they do support the explicit specification of preference relations. In fact, we may even conduct comparisons between PLP and `dlv` or `smodels` *directly*, by representing a problem equivalently with and without the use of ordering relations, and so being able to evaluate the representational power of explicit preferences.

Problem Size

`dlv` and `smodels` are state-of-the-art implementations successfully demonstrating their respective ability to process large and complex problem descriptions. Because our compiler is an efficient translator (the resultant logic programs are polynomial in the size of the input program), PLP likewise scales up to real-life problem representations.

References

[Baader & Hollunder 1993] Baader, F., and Hollunder, B. 1993. How to prefer more specific defaults in terminological default logic. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 669–674.

[Brewka & Eiter 1999] Brewka, G., and Eiter, T. 1999. Preferred answer sets for extended logic programs. *Artificial Intelligence* 109(1-2):297–356.

[Brewka 1994] Brewka, G. 1994. Adding priorities and specificity to default logic. In Pereira, L., and Pearce, D., eds., *European Workshop on Logics in Artificial Intelligence (JELIA'94)*, Lecture Notes in Artificial Intelligence, 247–260. Springer Verlag.

[Brewka 1996] Brewka, G. 1996. Well-founded semantics for extended logic programs with dynamic preferences. *Journal of Artificial Intelligence Research* 4:19–36.

[Cholewiński *et al.* 1995] Cholewiński, P.; Marek, V.; Mikitiuk, A.; and Truszczyński, M. 1995. Experimenting with nonmonotonic reasoning. In *Proceedings of the International Conference on Logic Programming.* MIT Press.

[Delgrande & Schaub 1997] Delgrande, J., and Schaub, T. 1997. Compiling reasoning with and about preferences into default logic. In Pollack, M., ed., *Proceedings of the International Joint Conference on Artificial Intelligence*, 168–174. Morgan Kaufmann Publishers.

[Delgrande, Schaub, & Tompits 2000] Delgrande, J.; Schaub, T.; and Tompits, H. 2000. Logic programs with compiled preferences. In *Proceedings of the Eight International Workshop on Nonmonotonic Reasoning.*

[Eiter *et al.* 1997] Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1997. A deductive system for nonmonotonic reasoning. In Dix, J.; Furbach, U.; and Nerode, A., eds., *Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning*, volume 1265 of *Lecture Notes in Artificial Intelligence*, 363–374. Springer Verlag.

[Eiter *et al.* 1999] Eiter, T.; Faber, W.; Leone, N.; and Pfeifer, G. 1999. The diagnosis frontend of the `dlv` system. *AI Communications* 12(1–2):99–111.

[Eiter *et al.* 2000] Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2000. Using the dlv system for planning and diagnostic reasoning. In *Proceedings 14th Workshop on Logic Programming (WLP '99)*, 125–134. GMD.

[Gelfond & Lifschitz 1988] Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the International Conference on Logic Programming*, 1070–1080. MIT Press.

[Gelfond & Son 1997] Gelfond, M., and Son, T. 1997. Reasoning with prioritized defaults. In Dix, J.; Pereira, L.; and Przymusinski, T., eds., *Third International Workshop on Logic Programming and Knowledge Representation*, volume 1471 of *Lecture Notes in Computer Science*, 164–223. Springer Verlag.

[Niemelä & Simons 1997] Niemelä, I., and Simons, P. 1997. Smodels: An implementation of the stable model and well-founded semantics for normal logic programs. In Dix, J.; Furbach, U.; and Nerode, A., eds., *Proceedings of the Fourth International Conference on Logic Programing and Nonmonotonic Reasoning*, 420–429. Springer.

[Zhang & Foo 1997] Zhang, Y., and Foo, N. 1997. Answer sets for prioritized logic programs. In Maluszynski, J., ed., *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, 69–84. MIT Press.