# plp

## A compiler for logic programs with preferences

James Delgrande Torsten Schaub Hans Tompits

5th December 2000

## 1 Introduction

These pages describe a system for compiling ordered logic programs into regular logic programs under answer set semantics [6]. Our system is conceived as a front-end to the logic programming systems `dlv` and `smodels`.

The underlying compiler is implemented in the programming language Prolog; it has been developed under the ECLiPSe Constraint Logic Programming System but its code employs standard Prolog programming constructs so that it remains portable to other Prolog systems. For instance, it runs equally well under SICStus Prolog.

We deal with extended logic programs under the answer set semantics [6] which is an extension of the stable models semantics [5] for handling logic programs with classical negation. An excellent introduction to logic programming under these semantics is due to Vladimir Lifschitz and can be accessed through his home-page.

This emerging subfield common to logic programming and nonmonotonic reasoning is also referred to *Answer Set Programming.* Some links on the subject are given here.

## 2 Beginner's guide

For those not familiar with preferences in logic programming, it might already be instructive to compare the results obtained on a simple example with and without preferences:

- Consider the well-known penguin-birds-example *without* preferences.

$$\begin{array}{rcl} penguin(tweety) & \leftarrow & \\ bird(tweety) & \leftarrow & \\ flies(tweety) & \leftarrow & bird(tweety), not\ \neg flies(tweety) \\ \neg flies(tweety) & \leftarrow & penguin(tweety), not\ flies(tweety) \end{array}$$

  This program is expressed in our syntax as follows:

```
    penguin(tweety).
    bird(tweety).
    flies(tweety) :- not neg flies(tweety), bird(tweety).
neg flies(tweety) :- not     flies(tweety), penguin(tweety).
```

Compiling the file, named `penguin.lp`, and piping the result into `dlv` yields two answer sets:

```
dlv [build BEN/Nov 24 1999   gcc egcs-2.91.66 19990314 (egcs-1.1.2 release)]

Reading "Examples/penguin.dlv"


{true, penguin(tweety), bird(tweety), flies(tweety)}

{true, penguin(tweety), bird(tweety), neg_flies(tweety)}
```

- Consider now the well-known penguin-birds-example *with* preferences.

  For this, we name the two last rules:

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

  Rules $r_1$ and $r_2$ are associated with names $n_1$ and $n_2$, which are constants in the object language. Fact $n_1 \prec n_2 \leftarrow$ expresses, intuitively, that $r_2$ has "higher priority" than $r_1$.

  This program 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.
```

  Compiling the file `penguin.lp` and piping the result into `dlv` yields now only one answer set:

```
dlv [build BEN/Nov 24 1999   gcc egcs-2.91.66 19990314 (egcs-1.1.2 release)]
```

```
Reading "Examples/penguin.dlv"


{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)}
```

Here we obtain a single answer set that corresponds to the second answer set above; it includes further atoms that stem from the additional preference-handling rules.

**Remarks.**

- The current version has been developed under ECLiPSe Prolog.

# 3 Getting started

The best way of getting started is to consult an exemplary session under ECLiPSe Prolog. An analogous session under SICStus Prolog is also available. In these session we took the above penguin-birds-example and proceeded in the following way:

1. We started by loading the compiler

   ```
   [eclipse 1]: [pref].
   ```

   This also charged the Prolog files `pp.pl`, `grounder.pl`, `dlv.pl`, `smodels.pl`. (see Section 10).

2. We compiled the original file penguin.lp by means

   ```
   [eclipse 2]: lp2dlv('Examples/penguin').
   ```

   This resulted in the file penguin.dlv that is then used as an input file to the `dlv` system in the next step.

   Our compiler is pretty verbose and displays also intermediate versions of the compiled program.

3. We finally call `dlv` (twice) by issuing the commands:

   ```
   [eclipse 3]: dlv('Examples/penguin').
   ```

   and

   ```
   [eclipse 4]: dlv('Examples/penguin',nice).
   ```

# 4 Syntax

## 4.1 Special purpose predicates

### 4.1.1 Input language

**Logical Symbols**

- true/0 is a predefined fact (that is used for dlv in order to make facts go into the intentional database);
- false/0 is a predefined symbol never to be found in any answer set (used for defining classical negation via integrity constraints);
- neg/1 (and -/1) are prefix predicates standing for classical negation;
- not/1 (and ~/1) are prefix predicates standing for negation as failure;
- ,/2 is conjunction;
- ;/2 (or v/2 or |/2) is disjunction;
- :-/2 is used for describing rules in the usual way.

**Predicates for handling preferences**

- name/1 is used for naming rules.

  This literal must occur as the *first* body literal, if present.

  Alternatively, one may use [mymostpreferredrule] instead of name(mymostpreferredrule).
- </2 is used for indicating preference among different rules (sets of rules).
- \</2 allows to express negated preference.

  That is, r \< s is equivalent to neg (r < s) .

### 4.1.2 Internal predicates

- prec/2 is used as the internal predicate indicating preference; it is thus the internal counterpart of </2.
- Predicate symbols starting with neg_ are reserved for (internally) expressing classical negation.
- Predicate symbols ap/1, bl/1, ok/1 and oko/2 are reserved for internal purposes.
- When dealing with set-ordered logic programs, as described in Section 8, we reserve moreover: nap/1, napo/2, nhead/1, ko/1, in/2 and sname/1.

### 4.2 Facts and rules

Facts and rules are formalized in the usual Prolog-ish way, with a few exceptions:

- usage of naming predicate `name/1` is restricted to the first body literal;
- facts cannot be named;
- predicates for dealing with preferences are subject to compilations and must thus not be used in the input language;
- internal predicates must not be used in the input language.

For examples see Section 6.

# 5 Command predicates

**General translations:**

- `lp2pl/1`

  Takes a Filename `<filename>` and compiles file `<filename>.lp` into file `<filename>.pl`.

**System specific translations:**

- `lp2dlv/1`

  Takes a filename `<filename>` and compiles file `<filename>.lp` into file `<filename>.dlv`, which is directly usable by `dlv`.

- `lp2sm/1`

  Takes a filename `<filename>` and compiles file `<filename>.lp` into file `<filename>.sm`, which is directly usable by `smodels`.

- `dlv/1`

  Takes a filename `<filename>` and pipes the file `<filename>.dlv` into `dlv`.

  There is a binary version `dlv/2` that allows for passing options to `dlv`.

  Try `dlv(<filename>,nice)`.

- `smodels/1`

  Takes a filename `<filename>` and pipes the file `<filename>.sm` into `smodels`.

# 6 Dynamic Preferences

All of the above applies to dynamic preferences.

## 6.1 Examples

The following table contains a list of examples discussed in the literature.

| Example | Source | DLV Code | Smodels Code |
|---|---|---|---|
| Delgrande, Schaub, and Tompits [4, ] | example.lp | example.dlv | example.sm |
| Folklore | penguin.lp | penguin.dlv | penguin.sm |
| Brewka and Eiter [2, Example 5.1] | be51.lp | be51.dlv | be51.sm |
| Brewka and Eiter [2, Example 5.2] | be52.lp | be52.dlv | be52.sm |
| Brewka and Eiter [2, Example 5.3] | be53.lp | be53.dlv | be53.sm |
| Brewka and Eiter [2, Example 5.4] | be54.lp | be54.dlv | be54.sm |
| Brewka and Eiter [2, Example 5.5] | be55.lp | be55.dlv | be55.sm |
| Baader and Hollunder [1, ] | bh.lp | bh.dlv | bh.sm |
| Baader and Hollunder [1, ] | bh2.lp | bh2.dlv | bh2.sm |
| Baader and Hollunder [1, ] | bh3.lp | bh3.dlv | bh3.sm |
| Gelfond and Son [7, Example 3.4] | gs34.lp | gs34.dlv | gs34.sm |
| Gelfond and Son [7, Example 3.4'] | gs34P.lp | gs34P.dlv | gs34P.sm |
| Gordon [8, chapter 2] | legal.lp | legal.dlv | legal.sm |
| Junker [9, running example] | junker.lp | junker.dlv | junker.sm |
| Delgrande, Schaub, and Tompits [4, ] | nonexP.lp | nonexP.dlv | nonexP.sm |
| Delgrande, Schaub, and Tompits [4, ] | nonexJ.lp | nonexJ.dlv | nonexJ.sm |

More examples for this collection are highly welcome!

# 7 Dynamic Preferences with Variables

With variables, we can now express conditional, context-sensitive preferences:

```
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)).

These steps are included in the procedure vlp2pl/2 and accordingly in vlp2dlv/2 and vlp2sm/2. It is important to note that files containing logic programs with variables are supposed to have the extension .vlp !

## 7.1 Command predicates

- vlp2pl/1 is the "variable"-counterpart of lp2pl/1

  Takes a Filename <filename> and compiles file <filename>.vlp into file <filename>.pl.

  This creates intermediate files:

  – <filename>.glp a (naively) grounded version of <filename>.vlp
    This is currently the crucial step because we use a brute-forth grounding mechanism that leads an unnecessary blow up. [1]

  – <filename>.alp a complex-term-free version of <filename>.glp (see above).

- vlp2sm/1 is the "variable"-counterpart of lp2sm/1

- vlp2dlv/1 is the "variable"-counterpart of lp2dlv/1

## 7.2 Examples

| Example | Source | DLV Code | Smodels Code |
|---|---|---|---|
| Folklore | penguin.lp | penguin.dlv | penguin.sm |
| Folklore | penguin1.lp | penguin1.dlv | penguin1.sm |
| Brewka and Eiter [2, 1st Example Chapter 8] | beQDM1.lp | beQDM1.dlv | beQDM1.sm |
| Brewka and Eiter [2, 2nd Example Chapter 8] | beQDM2.lp | beQDM2.dlv | beQDM2.sm |
| Gordon [8, chapter 2] | legal.lp | legal.dlv | legal.sm |

[1] We would like to use existing grounding procedures, like lparse, but we were so far afraid of the implicit program transformations.

# 8 Set Preferences

**Attention**: This is not a stand-alone file but rather a patch that overrides definitions in the standard compiler. It is thus necessary to recharge the file `pref.pl` whenever one wants to compile regular ordered logic programs after having dealt with sets of preferences.

## 8.1 What is the idea behind sets of preferences?

The idea is nicely illustrated with 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 in our syntax as follows, taken from file `car.lp`

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

% false        :- expensive, powerful, 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.
```

## 8.2 Expressing sets

Sets are expressed in an extensional way by:

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

Preferences on individual rules are expressed via the corresponding singleton sets.

## 8.3 Command predicates

We use the same commands as in 5 for sets of preferences.

### 8.4 Examples

The following table contains a list of examples discussed in the literature.

| Example | Source | DLV Code | Smodels Code |
|---|---|---|---|
| Delgrande and Schaub [3, running example] | car.lp | car.dlv | car.sm |
| - | ex0.lp | ex0.dlv | ex0.sm |
| - | ex1.lp | ex1.dlv | ex1.sm |
| - | nonGround.lp | nonGround.dlv | nonGround.sm |

More examples for this collection are highly welcome!

## 9 Compiling other approaches

### 9.1 Preferences à la Brewka and Eiter

Gerhard Brewka and Thomas Eiter have introduced in [2] another approach to preference handling. This approach is also implementable in our framework.

In common with previous work of Gerd Brewka, they begin with a partial order on a rule base, but define preference with respect to total orders that conform to the original partial order. As well, answer sets are first generated and the "preferred" answer sets are selected subsequently. To this end, they only deal with "static" preferences whose preferred answer sets are defined in terms of a modified definition of answer sets.

Our (prototypical) implementation of their approach is obtainable here. It is best tested with the examples given in Section 6.1.

### 9.2 Preferences à la Wang, Zhou, and Lin

Kewen Wang and collegues have introduced in [10] another approach to (static) preference handling. This approach is also implementable in our framework.

Our (prototypical) implementation of their approach is obtainable here. It is best tested with the examples given in Section 6.1.

## 10 The files

The compiler comes in 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 purpose of the file `pp.pl` is to provide a pretty-printer for the files resulting from the compilation.

  This module aims at transforming Prolog variables like `_4711` into more readable ones like `X`. This is necessary in view of processing the files with `dlv` and `smodels`.

- Files `dlv.pl` and `smodels.pl` contains code specific to the logic programming systems.

## 10.1 Source code

Here is the list of necessary files:

- `pref.pl`
- `pp.pl`
- `grounder.pl`
- `prefS.pl`
- `dlv.pl`
- `smodels.pl`

## 10.2 Documentation

- Here is a dvi version of this file.
- Here is a PostScript version of this file.

# 11 Versions of `dlv` and `smodels`

- We use `dlv` from November 24 , 1999.
- We use (since June 2000) `smodels version 2.25` and `lparse version 0.99.53`.

# 12 What's new?

**June 2000** TS started this history.

- added support for disjunctive logic programming; disjunctions can be stated in the head of rules by `v/2`, `;/2`, and `|/2`.

  This works only in conjunction with `dlv`.

- added support for smodels Version 2 and lparse

- dropped support for smodels Version 1 and parse
- added flag for switching on/off verbose mode; the flag is switched by `verbose_mode/0` and `no_verbose_mode/0`

**July 2000**

- removed ECLiPSe-specific features in pretty printer (`pp.pl`)
- added support for SICStus Prolog, along with Prolog system specific code
- added command predicates `lp4sm/1`, `lp4sm/2`, along with `lp4dlv/1`, `lp4dlv/2`, combining eg. `lp2sm/1` with `sm/1`

**December 2000**

- added module for translating the approach of [10]

# 13 Troubleshooting

- Be sure to have set a sufficient `print_depth` (also called `max_depth`).

  In ECLiPSe this can be done by means of comand: `set_flag(print_depth,100).`

- There are different initializations necessary, depending on whether one uses ECLiPSe or SICStus Prolog (see header of file `pref.pl`.).

  This is needed for using the `verbose_mode` flag.

# 14 Future gimmicks

- Different orderings.
- Better hiding of internal predicate symbols.
- Better documentation.

# References


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

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

[3] J. Delgrande and T. Schaub. Reasoning with sets of preferences in default logic. In H.-Y. Lee and H. Motoda, editors, *Pacific Rim International Conference on Artificial Intelligence*, volume 1531 of *Lecture Notes in Artificial Intelligence*, pages 134–145. Springer Verlag, 1998.

[4] J. Delgrande, T. Schaub, and H. Tompits. Logic programs with compiled preferences. In C. Baral and M. Truszczyński, editors, *Proceedings of the Eighth International Workshop on Non-Monotonic Reasoning*. `arXiv.org` e-Print archive, 2000.

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

[6] M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generation Computing*, 9:365–385, 1991.

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

[8] T. Gordon. *The pleading game: An Artificial Intelligence Model of Procedural Justice.* Dissertation, Technische Hochschule Darmstadt, Alexanderstraße 10, D-64283 Darmstadt, Germany, 1993.

[9] U. Junker. A cumulative model semantics for dynamic preferences on assumptions. In M. Pollack, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 162–167. Morgan Kaufmann Publishers, 1997.

[10] K. Wang, L. Zhou, and F. Lin. Alternating fixpoint theory for logic programs with priority. In *Proceedings of the First International Conference on Computational Logic*. Springer Verlag, 2000.