nlp: A Compiler for Nested Logic Programming*

Vladimir Sarsakov[1], Torsten Schaub[1**], Hans Tompits[2], and Stefan Woltran[2]

[1] Institut für Informatik, Universität Potsdam, Postfach 90 03 27, D–14439 Potsdam,
[2] Institut für Informationssysteme, TU Wien, Favoritenstraße 9–11, A–1040 Wien,

Abstract. nlp is a compiler for nested logic programming under answer set semantics. It is designed as a front-end translating nested logic programs into disjunctive ones, whose answer sets are then computable by disjunctive logic programming systems, like dlv or gnt. nlp offers different translations: One is polynomial but necessitates the introduction of new atoms, another is exponential in the worst case but avoids extending the language. We report experimental results, comparing the translations on several classes of benchmark problems.

1 Nested Logic Programs and their Compilation

Nested logic programs allow for arbitrarily nested formulas in heads and bodies of logic program rules under answer set semantics [4]. Nested expressions can be formed using conjunction, disjunction, and the negation-as-failure operator in an unrestricted fashion. Previous results [4] show that nested logic programs can be transformed into standard (unnested) disjunctive logic programs in an elementary way, applying the negation-as-failure operator to body literals only. This is of great practical relevance since it allows us to evaluate nested logic programs by means of off-the-shelf disjunctive logic programming systems, like dlv [3] and gnt [2]. However, it turns out that this straightforward transformation results in an exponential blow-up in the worst-case, despite the fact that complexity results indicate a polynomial translation among both formalisms. In [5], we provide such a polynomial translations of nested logic programs into disjunctive ones. This translation introduces new atoms reflecting the structure of the original program; in the sequel, we refer to it as the *structural translation*. Likewise, we refer to the possibly exponential one as being *language-preserving*.

2 The nlp System

Both translations are implemented as a front-end to dlv [3] and gnt [2], representing state-of-the-art implementations for disjunctive logic programs under answer set semantics. The resulting compiler, called nlp, is publicly available at

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

* The first two authors were supported by DFG under grant SCHA 550/6, TP C. All authors acknowledge support within IST-2001-37004 project WASP.

** Affiliated with the School of Computing Science at Simon Fraser University, Canada.

```
p;not_t:-not q,q.
p;not_t:-not q,not s.
not_t:-not t.
:-t,not_t.
```

```
p;not_t:-not q,l1.
q;not_s:-l1.
l1:-q.
l1:-not s.
not_t:-not t.
:-t,not_t.
not_s:-not s.
:-s,not_s.
```

Fig. 1. Resulting (optimized) code for program Π under the **(a)** language-preserving translation (`example.dlp`); and **(b)** structural translation (`example.htl`).

The current implementation runs under SICStus and SWI Prolog and comprises roughly 1000 lines of code. In addition, it contains a number of optional improvements that allow for more compact code generation.

Consider program $\Pi = \{p \leftarrow \neg(q \vee \neg t) \wedge (q \vee \neg s)\}$, where '$\neg$' expresses negation as failure; it is expressed by `p :- not (q;not t), (q;not s).` Once a file is read into `nlp`, it is subject to multiple transformations. Most of these transformations are rule-centered in the sense that they apply in turn to each rule. While the original file is supposed to have the extension `nlp`, the result of the respective compilation is indicated by the extension `dlp` (language-preserving) and `htl` (structural), respectively. The system is best used with the command `nlp2`T`2`S for $T \in \{\texttt{dlp}, \texttt{htl}\}$ and $S \in \{\texttt{dlv}, \texttt{gnt}\}$; e.g., `nlp2htl2gnt(ex)` applies the structural translation (htl) to file `ex.nlp` and calls `gnt` on the result. Applying both transformations to our example Π yields the files in Figure 1a and 1b, respectively. The first rule in Figure 1a illustrates that we (currently) refrain from any sophisticated logical simplifications. On the other hand, the implemented versions of our translations are optionally improvable (via a flag) in two simple yet effective ways: First, no labels are generated for literals and second only a single label is generated for n-ary disjunctions or conjunctions. All our experiments are conducted using this optimization. Also, three different schemes for label generation are supported; here, we adhere to a number-oriented version (as seen in Figure 1b by `l1`).

3 Experimental Analysis

In view of the already diverging sizes on our toy example, it is interesting to compare the resulting images of our translations in a systematic way. To this end, we have implemented two different benchmark series that produce scalable problem instances.

Normal form series. The first one generates nested logic programs whose rule heads and bodies consist of two-level disjunctive and/or conjunctive normal form formulas. The generation of such programs is guided by 10 parameters: on the one hand, the number of variables, rules, facts, and constraints; and, on the other hand, the structure of heads and bodies is fixed by 3 parameters: the number of connectives on the first and second level plus the number of literals connected to

the first level. The placement of literals (p, $\neg p$, and $\neg\neg p$), respectively, is done randomly. These parameters are also reflected in the name of the source file. E.g., file `test_50.20_5_0.3_2_0.2_3_1.30.2.nlp` comprises a program over an alphabet of 50 variables, containing 20 rules, such as

```
p42,p4;p11,not p50;p46,p26 :-
        (not not p10;p37;p27),(p20;p34;p39),not not p11.
```

plus 5 facts and 0 constraints, expressed by `20_5_0`. While the primary connective of the head is a disjunction, namely '`;`', the one of the body is conjunction '`,`'. Although this can be arbitrarily fixed, it is currently set this way in order to provoke a significant blow-up when translating programs by means of successive applications of distributivity. The above head parameters, '`3_2_0`', enforce heads (being disjunctions) composed of 3 conjunctions, containing 2 literals, along with zero literal disjuncts. The body parameters, '`2_3_1`', lead to conjunctions with 2 disjunctions, containing 3 literals, and 1 literal conjunct, viz. '`not not p11`' in the rule above. Finally, the filename ends with an estimated blow-up factor for the language-preserving translation, viz. '`30.2`' in our example.

Given a nested program Π, its blow-up factor, $\theta(\Pi)$, is computed as follows:

$$\theta(\Pi) = \frac{\left[r\cdot(u_1+u_3+v_1+v_3)\cdot v_2^{v_1}\cdot u_2^{u_1}\right]+\left[h\cdot(u_1+u_3)\cdot u_2^{u_1}\right]+\left[b\cdot(v_1+v_3)\cdot v_2^{v_1}\right]}{[(r+h)\cdot(u_1\cdot u_2+u_3)]+[(r+b)\cdot(v_1\cdot v_2+v_3)]},$$

where r stands for the number of rules; h, b gives the number of heads and bodies in the program; u_1, u_2, u_3 are the head parameters; and v_1, v_2, v_3 are the body parameters. The denominator of $\theta(\Pi)$ gives the size of Π in terms of its number of literals, whilst the numerator captures the size of the disjunctive program resulting from the language-preserving translation.

Cardinality constraints series. The second generator is based on the idea that (single headed) logic programs with cardinality literals can be transformed into nested ones [1]. A cardinality literal of form $m\{L_1, \ldots, L_n\}n$ means informally that the resulting answer set must contain at least m and at most n literals out of $\{L_1, \ldots, L_n\}$; see [6] for a semantics. Such literals can then be used in the same way as ordinary literals in the head as well as the body of rules. As above, we start with generating a parameterized program with cardinality literals. For brevity, we omit further details and refer the interested reader to the URL below.

Experiments. The generators and test series can be downloaded at

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

In all, we generated 4661 programs, all of which where translated by each of the translations and the respective image was subsequently passed to `dlv`. All tests were run under Linux (Mandrake 8.1) on bi-processors based on AMD Athlon MP 1200 MHz with 1GByte main memory. For simplicity, we fixed the initial size of the programs: While in the first series all nested programs contain 20 rules and 5 facts, the same holds in the second series regarding cardinality constraints. For simplicity, both series are further divided into subseries according to the number of variables involved; in fact, we ran subseries with 10, 15, 20,. . . , 45, 50 variables.

Since the results of all these subseries are reported at the aforecited URL, we concentrate here on one representative, given by the 25 variables subseries. This subseries comprises 260 normal form programs and 426 cardinality constraints

Table 1. Quantitative comparison.

	Normal form		Cardinality constraints	
Feature	dlp<htl	htl<dlp	dlp<htl	htl<dlp
Size in Bytes	15	245	146	279
Size in Rules	22	238	208	217
Compilation time	10	250	182	243
Runtime	63	190	282	115

programs. The corresponding plots are given at the above URL at `node20.html` and `node66.html`. In all cases, we observe, as expected, a significant increase in the size of the image under the language-preserving translation. This is confirmed in the experiments through the ratios between the size of the output and input file. Also, looking at Table 1 (where $x < y$ indicates how many times x is "larger" than y), we see that the majority of test cases has a smaller image under the structural transformations than under the language-preserving one. However, this is less significant for the cardinality constraints series than for the normal form series. Clearly, the size of the image affects also the compilation time. We thus get a similar behavior as regards compilation time, as we obtained for the size of the image (cf. Table 1). Things become more interesting when comparing the respective running times (for computing *all* answer sets of a given program; cf. the aforecited URLs). For the first time, we obtain a kind of threshold beyond which the structural transformations always outperform the language-preserving one. While this is the case for an estimated blow-up factor of 500 for the normal-form test cases, the one for the cardinality-constraint test cases is at 200. The fact that the results of all translations appear to be scattered below this threshold is confirmed by the detailed results under the threshold. Interestingly, the language-preserving translation may even be outperformed on rather small blow-up factors, as witnessed by the range of 0–50 (cf. again the above given URLs). Although these thresholds are sometimes less clear cut in the other test series, their existence seems to be a recurring feature.

References

1. P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 2003. To appear.
2. T. Janhunen, I. Niemelä, P. Simons, J. You. Unfolding partiality and disjunctions in stable model semantics. In *Proc. KR-00*, pages 411–419. Morgan Kaufmann, 2000.
3. N. Leone et al. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 2003. To appear.
4. V. Lifschitz, L. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.
5. D. Pearce, V. Sarsakov, T. Schaub, H. Tompits, S. Woltran. A polynomial translation of logic programs with nested expressions into disjunctive logic programs. In *Proc. ICLP-02*, pages 405–420. Springer, 2002.
6. P. Simons, I. Niemelä, T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.