A system with template answer set programs*

Francesco Calimeri, Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro

Department of Mathematics, University of Calabria - 87030 Rende (CS), Italy
{calimeri,ianni,ielpa,pietramala,santoro}@mat.unical.it

Abstract. Although ASP systems have been extended in many directions, they still miss features which may be helpful towards industrial applications, like capabilities of quickly introduce new predefined constructs or to deal with compound data structures and module. We show here an implementation on top of the DLV system of DLP^T language, which features increased declarativity, code readability, compactness and reusability.

1 Introduction

ASP has recently found a number of promising applications, like information integration and knowledge management (even in some projects funded by the European Commission [14, 13]). Indeed, the ASP community has produced several extensions of non-monotonic logic languages, aimed at improving readability and easy programming; in order to specify classes of constraints, search spaces, data structures, new forms of reasoning, new special predicates [1, 6, 15, 3, 2, 7].

We describe here the DLP^T system as an extension of ASP with template constructs. ASP systems developers are enabled to fast prototype, making new features quickly available to the community, and later to concentrate on efficient (long lasting) implementations. Template predicates allow to define intensional predicates by means of generic, reusable subprograms, easing coding and improving readability and compactness. For instance, a template program is like

```
#template max[p(1)](1) {
    exceeded(X) :- p(X),p(Y), Y > X.
    max(X) :- p(X), not exceeded(X). }
```

The statement above defines the predicate `max`, which computes the maximum value over the domain of a generic unary predicate `p`. A template definition may be instantiated as many times as necessary, through *template atoms* (or *template invocations*), like in `:-max[weight(*)](M),M>100.` Template definitions may be unified with a template atom in many ways. The above rule contains a *plain* invocation, while in `:-max[student(Sex,$,*)](M),M>25.` there is a *compound* one.

The DLP^T language has been successfully implemented and tested on top of the DLV system [9]. Anyway, the proposed paradigm does not rely at all on DLV special features, and is easily generalizable.

* This work was partially supported by the European Commission under projects IST-2002-33570 INFOMIX, and IST-2001-37004 WASP.

2 Syntax

A DLP^T program is an ASP program[1] containing (possibly negated) *template atoms*. A template definition D consists of two parts; *(i)* a template header, #**template** $n_D[f_1(b_1), \ldots, f_n(b_n)](b_{n+1})$, where each $b_i (1 \leq i \leq n+1)$ is a nonnegative integer value, $f_1, \ldots, f_n$ are predicate names (called *formal predicates*), and n_D is called *template name*; *(ii)* an associated DLP^T subprogram enclosed in curly braces; n_D may be used within the subprogram as predicate of arity b_{n+1}, whereas each predicate $f_i (1 \leq i \leq n)$ is intended to be of arity b_i. At least a rule having n_D in the head must be declared. For instance, the following (defining subsets of the domain of a given predicate `p`) is a valid template definition: `#template subset[p(1)](1) { subset(X) v -subset(X) :- p(X). }`.

A template atom t is of the form: $n_t[p_1(\mathbf{X}_1), \ldots, p_n(\mathbf{X}_n)](\mathbf{A})$, where $p_1, \ldots, p_n$ are predicate names (*actual* predicates), and n_t a template name. Each $\mathbf{X}_i (1 \leq i \leq n)$ is a list of *special* terms. A special list of terms can contain either a variable name, a constant name, a '`$`' symbol (*projection term*) or a '`*`' symbol (*parameter term*). Variables and constants are *standard* terms. Each $p_i(\mathbf{X}_i)(1 \leq i \leq n)$ is called *special* atom. $\mathbf{A}$ is a list of *standard* terms called *output list*. Given a template atom t, let $D(t)$ be the corresponding template definition. It is assumed there is a unique definition for each template name.

Briefly, projection terms ('`$`' symbols) indicate which attributes of an actual predicate have to be ignored. A *standard* term within an actual atom indicates a 'group-by' attribute, whereas parameter terms ('`*`' symbols) indicate attributes to be considered as parameter. An example of template atom is `max[company($,State,*)](Income)`. Intuitively, the extension of this predicate consists of the companies with maximum value of the `Income` attribute (the third attribute of the *company* predicate), grouped by `State` (the second attribute), ignoring the first attribute. The computed values of `Income` are returned through the output list.

3 Knowledge Representation

A couple of examples now follows. For instance it is possible to define aggregate predicates [16]. They allow to represent properties over sets of elements. The next template predicate counts distinct instances of a predicate `p`, given an order relation `succ` defined on the domain of `p`. Moreover, this definition does not suffer from semantic limitations [3] and can be invoked also in recursive components of the programs. We assume the domain of integers is bounded to some finite value.

```
#template count[p(1),succ(2)](1) {
    partialCount(0,0).
    partialCount(I,V) :- not p(Y), I=Y+1, partialCount(Y,V).
    partialCount(I,V2) :- p(Y), I=Y+1, partialCount(Y,V), succ(V,V2).
    partialCount(I,V2) :- p(Y),I=Y+1, partialCount(Y,V), max[succ(*,$)](V2).
    count(M) :- max[partialCount($,*)](M). }
```

A ground atom `partialCount(i,a)` means that, at the stage `i`, `a` has been counted up; `count` takes the value counted at the highest (i.e. the last) stage value.

It is worth noting how `max` is employed over the `partialCount` predicate, which is binary. The last rule is equivalent to the piece of code:

```
partialCount'(X) :- partialCount(_,X).
count(M) :- max[partialCount'(*)](M).
```

Templates may help introducing and reusing definitions of common search spaces.

[1] We assume the reader to be familiar with basic notions concerning with ASP syntax and semantics; for further information please refer to [5].

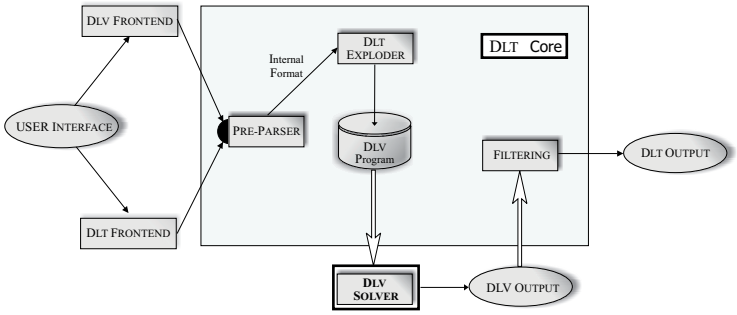

Fig. 1. Architecture of DLT system

```
#template permutation[p(1)](2). {
    permutation(X,N) v npermutation(X,N) :- p(X),#int(N),count[p(*),>(*,*)](N1),N<=N1.
    :- permutation(X,A),permutation(Z,A), Z <> X.
    :- permutation(X,A),permutation(X,B), A <> B.
    covered(X) :- permutation(X,A).
    :- p(X), not covered(X). }
```

Such kind of constructs enriching plain Datalog have been proposed, for instance, in [11, 1]. The above predicate ranges permutations over the domain of a given predicate `p`. In this case a ground atom `permutation(x,i)` tells that the element `x` (taken from the domain of `p`), is at position `i` within the currently guessed permutation. The rest of the template subprogram forces permutations properties to be met.

4 Informal Semantics

Semantics are given through a suitable ''explosion'' algorithm. Given a DLP^T program P, the *Explode* algorithm replaces each template atom t with a standard atom, referring to a fresh intensional predicate p_t. The subprogram d_t (which may have associated more than one template atom), defining the predicate p_t, is computed according to the template definition $D(t)$. The final output of the algorithm is a standard ASP program P'. Answer sets of the originating program P are constructed, *by definition*, from answer sets of P'. A full description of the ''explosion'' algorithm as well as many more details are available in [12].

5 System architecture and usage

The DLP^T language has been implemented on top of the `DLV` system [8–10], creating DLT system. The current version is available on the web [4, 12].

The overall architecture of the system is shown in Figure 1. The *Core* controls the whole process and interacts with frontend modules. A *Pre-parser* performs syntactic checks and builds an internal representation of the DLP^T program. The *Inflater* performs the *Explode* Algorithm and produces an equivalent `DLV` program P' which is piped towards the `DLV` system. The models $M(P')$ of P', computed by `DLV`, are then filtered out by the *Post-parser* in order to remove previously added internal information.

The `DLV` system is continuously enriched with new features; thus, the user is allowed to exploit the *ASITIS* directive in order to exclude from parsing some piece of code (containing constructs DLT is not aware of, but recognized by the underlying system). This allows to adapt DLT to other ASP systems with different syntax.

6 Current Work

We are working extending the framework *a)* generalizing template semantics for safe forms of recursion between invocations, *b)* introducing new forms of template atoms in order to improve reusability of the same template definition in different contexts, *c)* extending the template definition language using standard languages, such as `C++`.

Some experiments are being performed in order to have an idea about the overhead due to the "exploded" code. Encodings of well-known (e.g. *K-clique*, *hamiltonian path*, *3-colorability*, etc.) problems have been tested: "pure" ASP against *exploded* DLP^T ones (originally written exploiting templates). Overhead of the latter is never higher than 5%. However, performances are strictly tied to performances of resulting ASP programs; and it is worth remarking that this work, aiming at introducing fast prototyping techniques, does not consider time performances as a primary target[2].

References

1. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all the problems in NP. *Computer Languages, Elsevier Science, Amsterdam (Netherlands)*, 26(2-4):165–195, 2000.
2. W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, 1993.
3. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings IJCAI-2003*, Acapulco, Mexico, Aug. 2003.
4. The DLP^T web site. `http://dlpt.gibbi.com`.
5. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.
6. T. Eiter, G. Gottlob, and N. Leone. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.
7. T. Eiter, G. Gottlob, and H. Veith. Modular Logic Programming and Generalized Quantifiers. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of LPNMR-97*, number 1265 in LNCS, pages 290–309. Springer, 1997.
8. W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In *Proceedings of DDLP'99*.
9. W. Faber, N. Leone, and G. Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of IJCAI 2001*, pages 635–640, Seattle, WA, USA.
10. W. Faber and G. Pfeifer. `DLV` homepage, since 1996. http://www.dlvsystem.com/.
11. S. Greco and D. Saccà. NP optimization problems in datalog. *International Symposium on Logic Programming. Port Jefferson, NY, USA*, pages 181–195, 1997.
12. G. Ianni, F. Calimeri, G. Ielpa, A. Pietramala, and M. C. Santoro. Enhancing answer set programming with templates. In *Proceedings of NMR 2004*.
13. The ICONS web site. `http://www.icons.rodan.pl/`.
14. The Infomix web site. http://www.mat.unical.it/infomix.
15. G. M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41(1):44–64, 1990.
16. K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.

[2] We would like to thank Nicola Leone and Luigi Palopoli for their fruitful remarks.