Template Programs for Disjunctive Logic Programming: an Operational Semantics

Francesco Calimeri [a,*],
Giovambattista Ianni [a]

[a] *Department of Mathematics, University of Calabria*
87030 Rende(CS), Italy.
E-mail: {calimeri,ianni}@mat.unical.it

Disjunctive Logic Programming is nowadays a mature formalism which has been successfully applied to a variety of practical problems, such as information integration, knowledge representation, planning, diagnosis, optimization and configuration. Although current DLP systems have been extended in many directions, they still miss features which may be helpful towards industrial applications, like the capability of quickly introducing new predefined constructs or of dealing with modules. Indeed, in spite of the fact that a wide literature about modular logic programming is known, code reusability has never been considered as a critical point in Disjunctive Logic Programming. In this work we extend Disjunctive Logic Programming, under stable model semantics, with the notion of 'template' predicates. A template predicate may be instantiated to an ordinary predicate by means of template atoms, thus allowing to define reusable modules, to define new constructs and aggregates without any syntactic limitation.

Keywords: Disjunctive Logic Programming, Modular Logic Programming, Knowledge Representation

1. Introduction

Disjunctive Logic Programming (DLP) is nowadays a generic term including many language flavors, whose common base is the adoption of the 'Gelfond-Lifschitz reduct' [Gelfond and Lifschitz, 1991] as a main tool for defining the underlying semantics. Roughly speaking, Disjunctive Logic Programming dialects are variants of Datalog, where models for a given program (stable models) may be multiple. Most of these languages allow to filter out models by means of constraints or to select among different models by means of *weight* constraints or similar extensions [Buccafurri *et al.*, 2000; Niemelä *et al.*, 1999; Niemelä, 1999; Simons, 1999; Niemelä *et al.*, 1999; 2000; Marek and Remmel, 2002; Ferraris and Lifschitz, 2005].

After some pioneering work on stable model computation [Bell *et al.*, 1994; Subrahmanian *et al.*, 1995], research in the field produced several, mature, implemented systems featuring clear semantics and efficient program evaluation [Seipel and Thöne, 1994; Babovich, since 2002; Chen and Warren, 1996; Cholewiński *et al.*, 1996; Aravindan *et al.*, 1997; Rao *et al.*, 1997; McCain and Turner, 1998; Cholewiński *et al.*, 1999; Egly *et al.*, 2000; East and Truszczyński, 2000; Anger *et al.*, 2001; East and Truszczyński, 2001; Leone *et al.*, 2005b; Lierler and Maratea, 2004; Lin and Zhao, 2004; Janhunen and Niemelä, 2004; Lierler, 2005].

DLP under the stable model semantics has recently found a number of promising applications: several tasks in information integration and knowledge management require complex reasoning capabilities, which are explored, for instance, in the INFOMIX and ICONS projects (funded by the European Commission)[Leone *et al.*, 2005a; ICONS, since 2001].

It is very likely that this new generation of DLP applications require the introduction of repetitive pieces of standard code. Indeed, a major need of complex and huge DLP applications such as [Nogueira *et al.*, 2001] is dealing efficiently with large pieces of such a code and with complex data structures, more sophisticated than the simple, native DLP data types.

Indeed, the non-monotonic reasoning community has continuously produced, in the past, several extensions of nonmonotonic logic languages, aimed

[*] Corresponding author: Francesco Calimeri, Department of Mathematics, University of Calabria, 87030 Rende(CS), Italy.

at improving readability and easy programming through the introduction of new constructs, employed in order to specify classes of constraints, search spaces, data structures, new forms of reasoning, new special predicates [Cadoli *et al.*, 1999; Eiter *et al.*, 1997a; Kuper, 1990], such as aggregate predicates [Calimeri *et al.*, 2005].

Nonetheless, code reusability has never been considered as a priority in the Answer Set Programming/DLP field, despite the fact that modular logic programming has been widely studied in the general case [Bugliesi *et al.*, 1994; Eiter *et al.*, 1997b]. The language DLP^T we propose here has two purposes. First, DLP^T moves the DLP field towards industrial applications, where code reusability is a crucial issue. Second, DLP^T aims at minimizing developing times in DLP system prototyping. DLP systems developers wishing to introduce new constructs are enabled to fast prototype their languages, make their language features quickly available to the scientific community, and successively concentrate on efficient (and long lasting) implementations. To this end, it is necessary a sound specification language for new DLP constructs. DLP itself proves to fit very well for this purpose. The proposed framework introduces the concept of 'template' predicate, whose definition can be exploited whenever needed through binding to usual predicates.

Template predicates can be seen as a way to define intensional predicates by means of a subprogram, where the subprogram is generic and reusable. This eases coding and improves readability and compactness of DLP programs:

Example 1 The following template definition

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

introduces a generic template program, defining the predicate `max`, intended to compute 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*, like in the following sample program

```
:- max[weight(*)](M), M > 100.          (a)
:- max[student(Sex,$,*)](M), M >25.     (b)
```

Template definitions may be unified with a template atom in many ways. The above program contains two invocations: a *plain* invocation (`a`), and a *compound* invocation (`b`). The latter allows to employ the definition of the template predicate `max` on a ternary predicate, discarding the second attribute of `student`, and grouping by values of the first attribute. □

The operational semantics of the language is defined through a suitable algorithm (actually, a *pseudo*-algorithm, as we will see in Section 4) which is able to produce, from a set of nonrecursive template definitions and a DLP^T program, an equivalent DLP program. There are some important theoretical questions to be addressed, such as the termination of the algorithm, and the expressiveness of the DLP^T language. Indeed, we prove that it is guaranteed that DLP^T program encodings are as efficient as plain DLP encodings, since unfolded programs are just polynomially larger with respect to the originating program.

The DLP^T language has been successfully implemented and tested on top of the DLV system [Faber *et al.*, 2001]. Anyway, the proposed paradigm does not rely at all on DLV special features, and is easily generalizable. In sum, benefits of the DLP^T language are: improved declarativity and succinctness of the code; code reusability and possibility to collect templates within libraries; capability to quickly introduce new, predefined constructs; fast language prototyping.

The paper is structured as follows:

– Section 2 briefly gives syntax and semantics of DLP and syntax of the language DLP^T.
– The features of DLP^T are illustrated in Section 3, with the help of some examples.
– Section 4 formally introduces the semantics of DLP^T.
– Theoretical properties of DLP^T are discussed in Section 5.
– An implementation of the DLP^T language on top of a suitable DLP solver is presented in Section 6.
– Eventually, in Section 7, conclusions are drawn.

2. Syntax of the DLP^T language

We provide here the syntax of DLP^T. First, we give a survey on formal syntax and semantics of DLP.

2.1. Disjunctive Logic Programming

The flavor of DLP we will consider is basically consisting in Disjunctive Datalog enriched with *weak constraints*. For further background the reader can refer to [Eiter *et al.*, 2000; Gelfond and Lifschitz, 1991]. In addition, in [Apt and Bol, 1994; Dix, 1995] more comprehensive surveys on the semantics of disjunction and negation are given.

2.1.1. DLP *Syntax*
A (standard) *term* is either a variable or a constant. Usually, strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants, such as X and x, respectively.
An *atom* is an expression $p(t_1,\ldots,t_n)$, where p is a *predicate* of arity n and $t_1,\ldots,t_n$ are terms, such as $edge(a,X)$ or $p(X)$. A *classical literal* l is either an atom p (in this case, it is *positive*), or a (strongly) negated atom $\neg p$ (in this case, it is *negative*). A *negation as failure (NAF) literal* ℓ is of the form l or not l, where l is a classical literal; in the former case ℓ is *positive*, and in the latter case *negative*. Unless stated otherwise, by *literal* a NAF literal is meant.
Given a classical literal l, its *complementary* literal $\neg l$ is defined as $\neg p$ if $l = p$ and p if $l = \neg p$. A set L of literals is said to be *consistent* if, for every literal $l \in L$, its complementary literal is not contained in L.
A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \,\mathtt{v}\, \cdots \,\mathtt{v}\, a_n \leftarrow b_1, \cdots, b_k, \text{not } b_{k+1}, \cdots, \text{not } b_m \cdot (1)$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n$ is said to be the *head* of r, while the conjunction $b_1,\ldots,b_k$, not $b_{k+1},\ldots$, not b_m is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint.* A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule.* If the body of r is empty (i.e. $k = m = 0$), r is called *fact*, and usually the "$\leftarrow$" sign is omitted.
For any set L of classical literals, not $L = \{\text{not } l \mid l \in L\}$ is denoted. If r is a rule of form (1), then $H(r) = \{a_1,\ldots, a_n\}$ is the set of the literals in the head and $B(r)= B^+(r)\cup B^-(r)$ is the set of the body literals, where $B^+(r)$ (the *positive body*) is $\{b_1,\ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$.
A DLP *program* (alternatively, *disjunctive datalog program*) $\mathcal{P}$ is a finite set of rules. A not-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^-(r) = \emptyset$) is called *positive.* In positive programs negation as failure (not) does not occur, while strong negation ($\neg$) may be present. A $\mathtt{v}$-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : |H(r)| \leq 1$) is called *datalog program* (or *normal logic program*).
A term (an atom, a rule, a program, etc.) is *ground*, if no variable appears in it. A ground program is also called a *propositional* program.
Usually, we simply refer to programs, if we want to point out that they are not restricted to be positive, normal or ground.

Weak constraints. Weak constraints (see [Buccafurri *et al.*, 2000]) are defined as a variant of integrity constraints. In order to differentiate clearly between them, for weak constraints the symbol ':$\sim$' is adopted instead of '$\leftarrow$'. Additionally, a weight and a priority level (or layer) inducing a partial order of the weak constraints are specified explicitly.
Formally, a weak constraint wc is an expression of the form

$$:\sim \ b_1,\ldots,b_k, \ \text{not } b_{k+1},\ldots, \ \text{not } b_m \cdot \ [w:l]$$

where for $m \geq k \geq 0$, $b_1,\ldots,b_m$ are classical literals, while w (the *weight*) and l (the *level*, or *layer*) are positive integer constants or variables. For convenience, w and/or l might be omitted and are set to 1 in this case.
The sets $B(wc)$, $B^+(wc)$, and $B^-(wc)$ of a weak constraint wc are defined in the same way as for regular integrity constraints.

2.1.2. DLP *Semantics*
The most widely accepted semantics for DLP is the *Stable Model Semantics* proposed by Gelfond and Lifschitz in 1991. According to this semantics, a program may have several alternative stable models (but possibly none), each corresponding to a possible view of the world.

The semantics provided in this section is a generalization of the original semantics proposed for weak constraints in [Buccafurri *et al.*, 2000], as seen in [Leone *et al.*, 2005b].

Herbrand Universe. For any program $\mathcal{P}$, let $U_{\mathcal{P}}$ (the Herbrand Universe) be the set of all constants appearing in $\mathcal{P}$. In case no constant appears in $\mathcal{P}$, an arbitrary constant ψ is added to $U_{\mathcal{P}}$.

Herbrand Literal Base. For any program $\mathcal{P}$, let $B_{\mathcal{P}}$ be the set of all ground (classical) literals constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_{\mathcal{P}}$ (note that, for each atom p, $B_{\mathcal{P}}$ contains also the strongly negated literal $\neg p$).

Ground Instantiation. For any rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$. In a similar way, given a weak constraint w, $Ground(w)$ denotes the set of weak constraints obtained by applying all possible substitutions σ from the variables in w to elements of $U_{\mathcal{P}}$. For any program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the set $GroundRules(\mathcal{P}) \cup GroundWC(\mathcal{P})$, where

$$GroundRules(\mathcal{P}) = \bigcup_{r \in Rules(\mathcal{P})} Ground(r)$$

and

$$GroundWC(\mathcal{P}) = \bigcup_{w \in WC(\mathcal{P})} Ground(w)\cdot$$

For propositional programs, $\mathcal{P} = Ground(\mathcal{P})$ holds.

Stable Models. For every program $\mathcal{P}$, we define its stable models using its ground instantiation $Ground(\mathcal{P})$ in three steps: first we define the stable models of positive disjunctive datalog programs, then we give a reduction of disjunctive datalog programs containing negation as failure to positive ones and use it to define stable models of arbitrary disjunctive datalog programs, possibly containing negation as failure. Finally, we specify the way how weak constraints affect the semantics, defining the semantics of general programs.

An interpretation I is a set of ground classical literals, i.e. $I \subseteq B_{\mathcal{P}}$ w.r.t. a program $\mathcal{P}$. A consistent interpretation $I \subseteq B_{\mathcal{P}}$ is called *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive disjunctive datalog program), if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap I \neq \emptyset$ whenever $B(r) \subseteq I$. An interpretation $I \subseteq B_{\mathcal{P}}$ is a *stable model* for a positive disjunctive datalog program $\mathcal{P}$, if it is minimal (under set inclusion) among all interpretations that are closed under $\mathcal{P}$.[1]

Example 2 The positive program

$$\mathcal{P}_1 = \{a \lor \neg b \lor c\cdot\}$$

has the stable models $\{a\}$, $\{\neg b\}$, and $\{c\}$. Its extension

$$\mathcal{P}_2 = \{a \lor \neg b \lor c\cdot\,,\ \leftarrow a\cdot\}$$

has the stable models $\{\neg \mathtt{b}\}$ and $\{\mathtt{c}\}$. Finally, the positive program

$$\mathcal{P}_3 = \{a \lor \neg b \lor c\cdot\,,\ \leftarrow a\cdot\,,\ \neg b \leftarrow c\cdot\,,\ c \leftarrow \neg b\cdot\}$$

has the single stable model the set $\{\neg b, c\}$.

The *reduct* or *Gelfond-Lifschitz transform* of a ground program $\mathcal{P}$ w.r.t. a set $I \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^I$, obtained from $\mathcal{P}$ by

- deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap I \neq \emptyset$ holds;
- deleting the negative body from the remaining rules.

A stable model of a program $\mathcal{P}$ is a set $I \subseteq B_{\mathcal{P}}$ such that I is a stable model of $Ground(\mathcal{P})^I$.

Example 3 Given the general program $\mathcal{P}_4$:

$$\begin{aligned} a \lor \neg b &\leftarrow c\cdot \\ \neg b &\leftarrow \text{not } a, \text{not } c\cdot \\ a \lor c &\leftarrow \text{not } \neg b\cdot \end{aligned}$$

and the interpretation $I = \{\neg \mathtt{b}\}$, the reduct $\mathcal{P}_4^I$ is $\{a \lor \neg b \leftarrow c\cdot\,,\ \neg b\cdot\}$. It is easy to see that I is a stable model of $\mathcal{P}_4^I$, and for this reason it is also a stable model of $\mathcal{P}_4$.
Now consider the interpretation $J = \{\mathtt{a}\}$. The reduct $\mathcal{P}_4^J$ is $\{a \lor \neg b \leftarrow c\cdot\,,\ a \lor c\cdot\}$ and it can be easily verified that J is a stable model of $\mathcal{P}_4^J$, so it is also stable model of $\mathcal{P}_4$.
If, on the other hand, we take $K = \{\mathtt{c}\}$, the reduct $\mathcal{P}_4^K$ is equal to $\mathcal{P}_4^J$, but K is not stable model of $\mathcal{P}_4^K$: for the rule $r : a \lor \neg b \leftarrow c$, $B(r) \subseteq K$ holds, but $H(r) \cap K \neq \emptyset$ does not. Indeed, it can be verified that I and J are the only stable models of $\mathcal{P}_4$.

[1]Note that we only consider *consistent stable models*, while in [Gelfond and Lifschitz, 1991] also the inconsistent set of all possible literals can be a valid stable models.

Given a ground program $\mathcal{P}$ with a set of weak constraints $WC(\mathcal{P})$, we are interested in the stable models of *Rules*$(\mathcal{P})$ which minimize the sum of weights of the violated (unsatisfied) weak constraints in the highest priority level[2], and among them those which minimize the sum of weights of the violated weak constraints in the next lower level, etc. Formally, this is expressed by an objective function $H^{\mathcal{P}}(A)$ for $\mathcal{P}$ and a stable model A as follows, using an auxiliary function $f_{\mathcal{P}}$ which maps leveled weights to weights without levels:

$$\begin{aligned}
&f_{\mathcal{P}}(1) = 1,\\
&f_{\mathcal{P}}(n) = f_{\mathcal{P}}(n-1) \cdot |WC(\mathcal{P})| \cdot w^{\mathcal{P}}_{max} + 1, \quad n > 1,\\
&H^{\mathcal{P}}(A) = \textstyle\sum_{i=1}^{l^{\mathcal{P}}_{max}} (f_{\mathcal{P}}(i) \cdot \sum_{w \in N^{\mathcal{P}}_i(A)} weight(w)),
\end{aligned}$$

where $w^{\mathcal{P}}_{max}$ and $l^{\mathcal{P}}_{max}$ denote the maximum weight and maximum level over the weak constraints in $\mathcal{P}$, respectively; $N^{\mathcal{P}}_i(A)$ denotes the set of the weak constraints in level i that are violated by A, and $weight(w)$ denotes the weight of the weak constraint w. Note that $|WC(\mathcal{P})| \cdot w^{\mathcal{P}}_{max} + 1$ is greater than the sum of all weights in the program, and therefore guaranteed to be greater than the sum of weights of any single level.
Intuitively, the function $f_{\mathcal{P}}$ handles priority levels. It guarantees that the violation of a single constraint of priority level i is more "expensive" then the violation of *all* weak constraints of the lower levels (i.e., all levels $< i$).
For a program $\mathcal{P}$ (possibly with weak constraints), a set A is an *(optimal) stable model* of $\mathcal{P}$ if and only if (1) A is a stable model of *Rules*$(\mathcal{P})$ and (2) $H^{\mathcal{P}}(A)$ is minimal over all the stable models of *Rules*$(\mathcal{P})$.

Example 4 Consider the following program $\mathcal{P}_{wc}$, which contains three weak constraints:

$$\begin{aligned}
&a \text{ v } b\cdot\\
&b \text{ v } c\cdot\\
&d \text{ v} \neg d \ \leftarrow \ a, c\cdot\\
&:\sim b\ \cdot \quad [1:2]\\
&:\sim a, \neg d\ \cdot \quad [4:1]\\
&:\sim c, d\ \cdot \quad [3:1]
\end{aligned}$$

Rules$(\mathcal{P}_{wc})$ admits three stable models: $A_1 = \{a, c, d\}$, $A_2 = \{a, c, \neg d\}$, and $A_3 = \{b\}$. We have: $H^{\mathcal{P}_{wc}}(A_1) = 3$, $H^{\mathcal{P}_{wc}}(A_2) = 4$, $H^{\mathcal{P}_{wc}}(A_3) = 13$.
Thus, the unique (optimal) stable model is $\{a, c, d\}$ with weight 3 in level 1 and weight 0 in level 2.

[2]Higher values for weights and priority levels mark weak constraints of higher importance. E.g., the most important constraints are those having the highest weight among those with the highest priority level.

2.2. DLP^T

A DLP^T program is a DLP program where (possibly negated) *template atoms* may appear in rules and constraints. Definition of template atoms is next provided.

Definition 1 A *template definition D* consists of:

- a template header,

  #template $n_D[f_1(b_1)\,, \cdots, f_n(b_n)](b_{n+1})$

  where $b_1, \ldots, b_{n+1}$ are (nonnegative) integer values, and $f_1, \ldots, f_n$ are predicate names (*formal predicates*, from now on). n_D is called *template name*;
- 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 the predicates $f_i, \ldots, f_n$ are intended to be of arity $b_i, \ldots, b_n$, respectively. At least a rule having n_D within its head must appear in the subprogram.

Example 5 Beside the one introduced in Example 1, another valid template definition is the following:

```
#template subset[p(1)](1)
{
subset(X) v  -subset(X) :- p(X).
}
```

Intuitively, this defines a subset of the predicate 'p'; such a subset is non-deterministically chosen by means of disjunction.

Definition 2 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 (namely, *actual* predicates), and n_t is a template name. $\mathbf{X}_i, \ldots, \mathbf{X}_n$ are lists of *special* terms (referred in the following as *special* lists of terms), where $\mathbf{A}$ is a list of standard terms.
A special term is either a standard term, or a dollar ('\$') symbol (from now on, *projection term*) or a star ('*') symbol (from now on, *parameter term*).
$p_1(\mathbf{X}_i), \ldots, p_n(\mathbf{X}_n)$ are called *special* atoms. $\mathbf{A}$ is called *output list*.
Given a template atom t, let $D(t)$ be the corresponding template definition having the same template name. It is assumed there is a unique definition for each template name.

Example 6 Some template atoms are

```
max[company($,State,*)](Income).
subset[node(*)](X).
```

Template atoms may "instantiate" template definitions as many times as necessary.

Example 7 The following short piece of program contains multiple instantiation of the 'max' template, whose definition has been introduced in Example 1:

```
:- max[weight(*)](M), M > 100.
:- max[student(Sex,$,*)](M), M > 25.
```

Looking at Example 6 and Example 7, we can get some intuitions on ('\$' and '*' symbols). Basically, projection terms ('\$' symbols) are intended to indicate which attributes, among those belonging to an actual predicate, have to be ignored. A standard term (a constant or a variable) within an actual atom indicates a 'group-by' attribute, whereas parameter terms ('*' symbols) indicate which attributes have to be considered as parameters.
Thus, the intuitive meaning of the first template atom of example 6 is to compute the companies with the maximum value of the 'income' attribute (the third attribute of the *company* predicate), grouped by the 'state' attribute (the second one), ignoring the first attribute. The computed values of *Income* are returned through the output list.

Example 8 Given a database by means of facts like

```
emp_companyA("Jones",30000,35,"Accounting").
 [...]
emp_companyB("Miller",34000,29,"Marketing").
```

the following single-rule program

```
emp_companyAB(Name) :-
  intersection[emp_companyA](*,$,$,$),
  emp_companyB(*,$,$,$)](Name).
```

computes the employees working for both company A and company B. It exploits the template 'intersection', defined in Section 3, and again shows how '\$' and '*' symbols can be used. The last three attributes (name, salary, department) are thus ignored, by meaning of '\$' symbols, while the first (name) is intended as parameter, by meaning of '*' symbol.

3. Knowledge Representation by DLP^T

In this section we show by examples the main advantages of template programming. Examples point out the provision of a succinct, elegant and easy-to-use way for quickly introducing new constructs through the DLP^T language.

Aggregates. Aggregate predicates [Ross and Sagiv, 1997], allow to represent properties over sets of elements. Aggregates or similar special predicates have been already studied and implemented in several DLP solvers [Dell'Armi *et al.*, 2003; Simons, 2000]: the next example shows how to fast prototype aggregate semantics without taking into account of the efficiency of a built-in implementation. Here we take advantage of the template predicate max, defined in Example 1. The next template predicate defines a general program to count distinct values of a predicate p, given an order relation succ defined on the domain of p. 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).
}
```

The above template definition is conceived in order to count, in a iterative-like way, values of the p predicate through the *partialCount* predicate. A ground atom *partialCount*(i, a) means that at the stage i, the constant a has been counted up.

The predicate *count* takes the value which has been counted at the highest (i.e. the last) stage value. The above program is somehow involved and shows how difficult could be to simulate aggregate constructs in Disjunctive Logic Programming. Anyway, the use of templates allows to write it once, and reuse it as many times as necessary.
It is worth noting how `max` is employed over the binary predicate `partialCount`, instead of an unary one. Indeed, the '`$`' and '`*`' symbols are employed to project out the first argument of `partialCount`. The last rule is equivalent to the piece of code:

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

Definition of ad hoc search spaces. Template definitions can be employed to introduce and reuse constructs defining the most common search spaces. This improves declarativity of DLP programs to a larger extent. The next two examples show how to define a predicate `subset` and a predicate `permutation`, ranging, respectively, over subsets and permutations of the domain of a given predicate `p`. Such kind of constructs enriching plain Datalog languages have been proposed, for instance, in [Laenens *et al.*, 1990; Cadoli and Schaerf, 2001].

```
#template subset[p(1)](1)
{
subset(X) v -subset(X) :- p(X).
}

#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).
}
```

The explanation of the `subset` template predicate (already appeared in Example 5 is quite straightforward. As for the `permutation` definition, a ground atom $permutation(x, i)$ states that the element x (taken from the domain of p), is in position i within the currently guessed permutation. The rest of the template subprogram forces permutations properties to be met.
Next we show how `count` and `subset` can be exploited to succinctly encode the *k-clique* problem [Garey and Johnson, 1979], i.e., given a graph G (represented by predicates `node` and `edge`), find if there exists a complete subgraph containing at least k nodes (we consider here the 5-clique problem):

```
in_clique(X) :- subset[node(*)](X).
:- count[in_clique(*),>(*,*)](K), K < 5.
:- in_clique(X),in_clique(Y), X <> Y,
  not edge(X,Y).
```

The first rule of this example guesses a clique from a subset of nodes. The first constraint forces a candidate clique to be at least of `5` nodes, while the last forces a candidate clique to be strongly connected. The `permutation` template can be employed, for instance, to encode the Hamiltonian Path problem: given a graph G, find a path visiting each node of G exactly once:

```
path(X,N) :- permutation[node(*)](X,N).
:- path(X,M), path(Y,N), not edge(X,Y),
  M = N+1.
```

The following `any` template may be employed in order to (non-deterministically) select exactly one value from the domain of a predicate p. It is built on top of the `subset` predicate.

```
#template any[p(1)](1)
{
any (X) :- subset[p(*)](X).
:- any(X), any(Y), X <> Y.
:- p(X), not any(X).
}
```

Handling complex data structures. DLP^T can be fruitfully employed to introduce operations over complex data structures, such as sets, dates, trees, etc.

Sets: Extending Datalog with Set programming is another matter of interest for the DLP field. This topic has been already discussed (e.g. in [Leone and Rullo, 1993; Kuper, 1990]), proposing some formalisms aiming at introducing a suitable semantics with sets. It is fairly quick to introduce set primitives using DLP^T; a set S is modeled through the domain of a given unary predicate s. Intuitive constructs like `intersection`, `union`, or `symmetricdifference`, can be modeled as follows.

```
#template intersection[a(1),b(1)](1).
{
intersection (X) :- a(X),b(X).
}

#template union[a(1),b(1)](1).
{
union(X) :- a(X).
union(X) :- b(X).
}

#template symmetricdifference[a(1),b(1)](1)
{
symmetricdifference(X) :- union[a(*),b(*)](X),
  not intersection[a(*),b(*)](X).
}
```

Dates: managing time and date data types is another important issue in engineering applications of DLP. For instance, in [Ianni *et al.*, 2003], it is very important to reason on compound records containing date values. The following template shows how to compare dates represented through a ternary relation ⟨day, month, year⟩.

```
#template before[date1(3),date2(3)](6)
{
before(D,M,Y,D1,M1,Y1) :- date1(D,M,Y),
  date2(D1,M1,Y1), Y<Y1.
before(D,M,Y,D1,M1,Y1) :- date1(D,M,Y),
  date2(D1,M1,Y1), Y==Y1, M<M1.
before(D,M,Y,D1,M1,Y1) :- date1(D,M,Y),
  date2(D,M1,Y1), Y==Y1,M==M1,D<D1.
}
```

4. Semantics of the DLP^T language

The semantics of the DLP^T language is given through a suitable "explosion" algorithm.

Remark 1 It is worth noting that, as already briefly mentioned, and more deeply discussed in Section 5, the "explosion" algorithm is actually a *pseudo*-algorithm, since it might not terminate in some cases. Nevertheless, we do prefer to keep the term *algorithm* also in the following.

It is given a DLP^T program P. The aim of the *Explode* algorithm, introduced next, is to remove template atoms from P. Each template atom t is replaced with a standard atom, referring to a fresh intensional predicate p_t. The subprogram d_t, defining the predicate p_t, is computed taking into account of the template definition $Def(t)$ associated to t. Actually, many template atoms may be grouped and associated to the same subprogram. The concept of atom signature, introduced next, helps in finding groups of equivalent template atoms. The final output of the algorithm is a DLP program P'. Stable models of the originating program P are constructed, *by definition*, from stable models of P'. Throughout this section, we will refer to Example 1 as running example. By little abuse of notation, $a \in P$ (resp. $a \in r$) means that the atom a appears in the program P (the rule r, respectively).

Definition 3 Given a template atom t, the corresponding *template signature* $s(t)$ is obtained from t by replacing each standard term with a conventional (mute variable) '_' symbol. Let $Def(s(t))$ be the template definition associated to the signature $s(t)$; Given a DLP^T program P, let $At(P)$ be the set of template atoms occurring in P. Let $Sig(At(P))$ be the set of signatures $\{s(t) : t \in At(P)\}$. □

For instance, `max[p(*,S,$)](M)` and `max[p(*,a,$)](H)` have the same signature, namely `max[p(*,_,$)](_)`.

4.1. The Explode algorithm

The **Explode** algorithm ($\mathcal{E}$ in the following) is sketched in Figure 4.1. It is given a DLP^{T}program P and a set of template definitions T. The output of $\mathcal{E}$ is a DLP program P'. $\mathcal{E}$ takes advantage of a stack of signatures S, which contains the set of signatures to be processed; S is initially filled up with each template signature occurring within P. The purpose of the main loop of $\mathcal{E}$ is to iteratively apply the $\mathcal{U}$ (Unfold) operation to P, until S is empty. Given a signature s, the $\mathcal{U}$ operation generates from the template definition $Def(s)$ a DLP^T program P^s which defines a fresh predicate t^s, where t is the template name of s. Then, P^s is appended to P; furthermore, each template atom $a \in P$, such that a has signature s, is replaced with a suitable atom $a^s(\mathbf{X}')$. It is important point-

```
Explode(Input: aDLP^T program P, a set of
            template definitions T.
        Outputs: an updated version of P'
            of P in DLP form.
        Data Structures: a queue S )
begin
        put each $s \in Sig(At(P))$ in S;
        $P' = P$;
        while ( S is not empty ) do begin
            extract a template signature s from S;

            //Start of the $\mathcal{U}$ (Unfold) operation;
              construct $P^s$ (see Subsection 4.2),
                    then set $P = P \cup P^s$;
              put all the $s' \in Sig(At(P^s))$ in S;
            for each template atom $a \in P$
                if a has signature s
                  construct the standard atom
                        $a^s(\mathbf{X}')$ (see Subsection 4.3);
                  replace a with $a^s(\mathbf{X}')$ in P;
            //End of the $\mathcal{U}$ operation;

        end;
end.
```

Fig. 1. The Explode ($\mathcal{E}$) Algorithm

ing out that, if P^s contains template atoms, the unfolding operation updates S with new template signatures.
We show next how P^s is constructed and template atoms are removed.
Let the header of $Def(s)$ be

$$\textbf{\#template}\ t[f_1(b_1)\,,\ \dots\ ,\ f_n(b_n)](b_{n+1})$$

Let s be of the form

$$t[p_1(\mathbf{X}_1)\,,\ \dots\ ,\ p_n(\mathbf{X}_n)](\mathbf{X}_{n+1})$$

Given a special list $\mathbf{X}$ of terms, let $\mathbf{X}[j]$ denote the j^{th} term of $\mathbf{X}$; let $fr(\mathbf{X})$ be a list of $|\mathbf{X}|$ fresh variables $F_{\mathbf{X},1}, \dots, F_{\mathbf{X},|\mathbf{X}|}$; let $st(\mathbf{X})$, $pr(\mathbf{X})$ and $pa(\mathbf{X})$ be the sublist of (respectively) standard, projection and parameter terms within $\mathbf{X}$. Given two lists $\mathbf{A}$ and $\mathbf{B}$, let $\mathbf{A}\&\mathbf{B}$ be the list obtained appending $\mathbf{B}$ to $\mathbf{A}$.

4.2. How P^s is constructed.

The program P^s is built in two steps. On the first step, P^s is enriched with a set of rules, intended in order to deal with projection variables.

For each $p_i \in s$, we introduce a predicate p_i^s and we enrich P^s with the auxiliary rule $p_i^s(\mathbf{X}_i') \leftarrow p_i(\mathbf{X}_i'')$, where:
- $\mathbf{X}_i''$ is built from $\mathbf{X}_i$ substituting $pr(\mathbf{X}_i)$ with $fr(pr(\mathbf{X}_i))$, substituting $pa(\mathbf{X}_i)$ with $fr(pa(\mathbf{X}_i))$, and substituting $st(\mathbf{X}_i)$ with $fr(st(\mathbf{X}_i))$;
- $\mathbf{X}_i'$ is set to $fr(st(\mathbf{X}_i))\&fr(pa(\mathbf{X}_i))$.

For instance, given the signature
$s_2 = max[student(,\$,*)](\_)$
and the example template definition given in Example 1, let $\mathbf{L}$ be the list $\langle$`_,$,*`$\rangle$; it is introduced the rule:

$$student^{s_2}(F_{st(\mathbf{L}),1}, F_{pa(\mathbf{L}),1})$$
$$:-student(F_{st(\mathbf{L}),1}, F_{pr(\mathbf{L}),1}, F_{pa(\mathbf{L}),1})\cdot$$

Note that projection variables are filtered out from $student^s$. In the second step, for each rule r belonging to $D(s)$, we create an updated version r' to be put in P^s, where each atom $a \in r$ is modified this way:
- if a is $f_i(\mathbf{Y})$ where f_i is a formal predicate, it is substituted with the atom $p_i^s(\mathbf{Y}')$. $\mathbf{Y}'$ is set to $fr(st(\mathbf{X}_i))\&\mathbf{Y}$;
- if a is a either a standard (included atoms having t as predicate name) or a special atom (in this latter case a occurs within a template atom) $p(\mathbf{Y})$, it is substituted with an atom $p^s(\mathbf{Y}')$, where
$Yvect' = fr(st(\mathbf{X}_1))\&\dots\&fr(st(\mathbf{X}_n))\&\mathbf{Y}\cdot$

Example 9 For instance, consider the rule
$max(X) \leftarrow\ p(X), not\ exceeded(X)\cdot$
from Example 1, and the signature
$s_2 =$ `max[student(_,$,*)](_)`;
let $\mathbf{L}$ be the special list $\langle \_, \$, * \rangle$; according to the steps introduced above, this rule is translated to

$$max^{s_2}(F_{\mathbf{L},1}, X) \leftarrow student^{s_2}(F_{\mathbf{L},1}, X),$$
$$not\ exceeded^{s_2}(F_{\mathbf{L},1}, X)\cdot\Box$$

4.3. How template atoms are replaced[3].

Consider a template atom in the form
$t[p_1(\mathbf{X}_1)\,,\ \dots\ ,\ p_n(\mathbf{X}_n)](\mathbf{X}_{n+1})\cdot$
It is substituted with
$t^s(\mathbf{X}')$
where
$\mathbf{X}' = st(\mathbf{X}_1)\&\dots\&st(\mathbf{X}_n)\&\mathbf{Y}.$

[3]Depending on the form of $D(s)$, some template atom might not to be allowed, since some atom with same predicate name but with mismatched arities could be generated.

Example 10 The complete output of $\mathcal{E}$ on the constraint

$$\leftarrow max[student(\_, \$, *)](M), M > 25\cdot$$

coupled with the template definition of `max` given in Example 1 is:

$$\begin{aligned}
student^{s_2}(S_1, P_1) \leftarrow\ & student(S_1, \_, P_1) \cdot \\
exceeded^{s_2}(F_{\mathbf{L},1}, X) \leftarrow\ & student^{s_2}(F_{\mathbf{L},1}, X), \\
& student^{s_2}(F_{\mathbf{L},1}, Y), \\
& Y > X \cdot \\
max^{s_2}(F_{\mathbf{L},1}, X) \leftarrow\ & student^{s_2}(F_{\mathbf{L},1}, X), \\
& not\ exceeded^{s_2}(F_{\mathbf{L},1}, X) \cdot \\
\leftarrow\ & max^{s_2}(Sex, M), M > 25\cdot
\end{aligned}$$

□

We are now able to give the formal semantics of DLP^T. It is important highlighting that stable models of a DLP^T program are, by definition, constructed in terms of stable models of an equivalent DLP program.

Definition 4 Given a DLP^T program P, and a set of template definitions T, let P' the output of the *Explode* algorithm on input $\langle P, T\rangle$. Let $H(P)$ be the Herbrand base of P' restricted to those atoms having predicate name appearing in P. Given a stable model $m \in M(P')$, then we define $H(P) \cap m$ as a stable model of P. □

Note that the Herbrand base of a DLP^T program is defined in terms of the Herbrand base of a DLP program *which is not* the output of $\mathcal{E}$.

5. Theoretical properties of DLP^T

The explosion algorithm replaces template atoms from a DLP^T program P, producing a DLP program P'. It is very important to investigate about two theoretical issues:
- Finding whether and when $\mathcal{E}$ terminates; in general, we observe that $\mathcal{E}$ might not terminate, for instance, in case of recursive template definitions. Anyway, we prove that it can be decided in polynomial time whether $\mathcal{E}$ terminates on a given input.
- Establishing whether DLP^T programs are encoded as efficiently as DLP programs. In particular, we are able to prove that P' is polynomially larger than P. Thus DLP^T keeps the same expressive power as DLP. This way, we are guaranteed that DLP^T program encodings are as efficient as plain DLP encodings, since unfolded programs are always reasonably larger with respect to the originating program.

Definition 5 It is given a DLP^T program P, and a set of template definitions T. The *dependency graph* $G_{T,P} = \langle V, E\rangle$ of T and P is a graph encoding dependencies between template atoms and template definitions, and it's built as follows. Each template definition $t \in T$ will be represented by a corresponding node u_t of V. V contains a node u_P associated to P as well. E will contain a direct edge $(u_t, u_{t'})$ if the template t contains a template atom referring to the template t' inside its subprogram (as for the node referred to P, we consider the whole program P). Let $G_{T,P}(u) \subseteq G_{T,P}$ be the subgraph containing nodes and arcs of $G_{T,P}$ reachable from u. □

Theorem 2 It is given a DLP^T program P, and a set of template definitions T. Let u_P the node of $G_{T,P}$ corresponding to P. If $G_{T,P}(u_P)$ is acyclic then $\mathcal{E}$ terminates whenever applied to P and T.
Proof. We assume $G_{T,P}(u_P) = \langle N, E\rangle$ is acyclic. we can state a partial ordering $\gg$ between its nodes, such that for each $v, v' \in N$, $v \gg v'$ iff either $(v, v') \in E$ or there is a v'' such that $v \gg v''$ and $v'' \gg v$.
We can build a total ordering $\succ$ by extending $\gg$ in a way that, whenever neither $v \gg v'$ nor $v \gg v'$ holds, it is chosen appropriately whether $v \succ v'$ or $v' \succ v$ holds. This can be done, by performing an in-depth visit of $G_{T,P}(u_P)$ and taking the resulting order of visit.
Let $level(v)$ be defined as follows:

- $level(v) = 0$ if there is no v' such that $v' \succ v$;
- for $i > 0$, $level(v) = i$ if i is the maximum value such that there is a v' such that $level(v') = i-1$ and $v \succ v'$.

Note that $level(v) > level(v')$ iff $v \succ v'$.
Given a queue S of signatures, let $level(S)$ be $\max_{Def(s)|s\in S} level(u_{Def(s)})$.

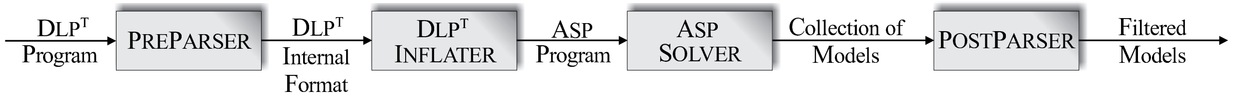

Fig. 2. Architecture of the DLP^T compiler

We will assume S is managed as a priority queue such that an element $s \in S$ having better value of $level(u_{Def(s)})$ is extracted first[4].

Note that $\mathcal{E}$ has a main loop where at each iteration a signature s is popped from S, whereas a new set of signatures S' is put in S. A new signature $s' \in S'$ can be put on S iff $Def(s) \succ Def(s')$. This means that $level(S)$ is non-increasing from one iteration to another.

$level(S)$ can stay unchanged from one iteration i to the next iteration $i+1$ only if there is some s' such that $Def(s) = Def(s')$ still in S at the beginning of iteration $i+1$. But, in this case, during iteration $i+1$, the cardinality of the set $\{s' \text{ s.t. } Def(s) = Def(s')\}$ is decreased by 1, since a new signature referring to the same template definition (and having same level) will be extracted from S.

Thus, there exists an iteration j, such that the difference $j - i$ has a maximum value bounded by $|\{s' \text{ s.t. } Def(s) = Def(s')\}|$, where s is the signature extracted at iteration i.

$\mathcal{E}$ will terminate once $level(S)$ is 0 and S is emptied up. □

Theorem 3 It is given a DLP^T program P, and a set of template definitions T. Let u_P the node of $G_{T,P}$ corresponding to P. If there is a cycle in $G_{T,P}(u_P)$ then $\mathcal{E}$ does not terminate.

Proof. In order to prove the theorem, we assume that there is a cycle $C = \{u_{t_0}, u_{t_1}, \ldots, u_{t_k}, u_{t_0}\}$, with $k \geq 0$ in $G_{T,P}(u_P) = \langle N, E\rangle$.

Since any node of N is reachable from u_P, we can also assume that $\mathcal{E}$ either loops infinitely or does not terminate until some node u_t such that $(u_t, u_{t_0}) \in E$ is reached, i.e. until $\mathcal{E}$ does not enters C or a similar cycle. This means that $\mathcal{E}$ will extract, during some iteration j, a signature s, such that $Def(s) = t$, from S, and then at least one s' such that $Def(s') = t_0$ and $s' \in Sig(At(P^s))$ is added to S.

We can prove that starting from the iteration j there is no iteration $j' > j$ such that $S = \emptyset$ at its beginning. j' can exist only if at the iteration $j'-1$ a remaining signature s_{last} is extracted and nothing else is added to S. Define S_C as the set of signatures such that $Def(s) \in \{t_0, \ldots, t_k\}$, that is, S_C is the set of signatures corresponding to nodes appearing in C. s_{last} cannot be member of S_C, because, in this case, an s_{last} will generate new signatures to be added in S. However, once C is reached, S will always contain, at the beginning of any iteration $j' > j$, at least one element of S_C. Indeed, it cannot be avoided that once an element $s' \in S_C$ is extracted, new elements of S_C are inserted during the iteration j'. □

[4]Although this assumption can be relaxed, we prefer to introduce it in order to keep the line of reasoning of this proof clearer.

Corollary 1 It is given a DLP^T program P, and a set of template definitions T. It can be decided in polynomial time whether $\mathcal{E}$ terminates when P and T are taken as input.

Proof. We observe that $G_{T,P}$ can be built in polynomial time; furthermore, by Theorem 2 we know that $\mathcal{E}$ terminates if $G_{T,P}(u_P)$ is acyclic, while, according to Theorem 3, it does not terminate if there is a cycle in $G_{T,P}(u_P)$. □

Definition 6 A set of template definitions T is said *nonrecursive* if for any DLP^T program P, the subgraph $G_{T,P}(u_P)$ is acyclic. □

It is useful to deal with nonrecursive sets of template definitions, since they may be safely employed with any program. Checking whether a set of template definitions is nonrecursive is quite easy.

Proposition 1 A set of template definitions T is nonrecursive iff $G_{T,\emptyset}$ is acyclic.

Proposition 2 Given a DLP^T program P and a nonrecursive set of template definitions T, the number of arcs of $G_{T,P}(u_P)$ is bounded by the overall size of T and P, i.e., it is $O(|T| + |P|)$.

Theorem 4 Given a DLP^T program P and a nonrecursive set of template definitions T, the output P' of $\mathcal{E}$ on input $\langle P, T\rangle$ is polynomially larger than P and T.

Proof. We first observe that each execution of $\mathcal{U}$ adds to P a number of rules (or constraints) whose overall size is clearly bounded by the size of T (see Figure 4.1). According to Theorem 2, if T is non-recursive, the number of $\mathcal{U}$ operations carried out by $\mathcal{E}$ is bounded by the maximum level l (bounded by the number of nodes of $G_{T,P}(u_P)$, and thus by the size of T) which can be assigned to a node of $G_{T,P}(u_P)$, times the number of different template atoms that occur in P and T. Thus, the size of P' is $O(|T|^2(|T| + |P|))$. □

In [Dantsin *et al.*, 2001] it's proved that plain DLP programs (under the brave reasoning semantics) entirely capture the complexity class Σ_2^P. This bounds the expressive power of DLP^T, too. Indeed, as previously shown, DLP^T programs may allow to express more succinct encodings of problems, w.r.t. DLP; but, despite this, the expressive power is not increased, accordingly to the following Corollary.

Corollary 2 DLP^T has the same expressive power as DLP.

Proof. The result is straightforward. Theorem 4 showed as unfolded DLP programs produced as the output of $\mathcal{E}$ are polynomially larger than the input programs. In addition, DLP^T semantics is defined in terms of the equivalent, unfolded, DLP program. Thus, DLP^T has the same expressiveness properties as DLP. □

6. System architecture and usage

The DLP^T language has been implemented on top of the DLV system [Faber *et al.*, 1999; 2001; Faber and Pfeifer, since 1996]. The current version of the language is available through the DLP^T Web page [Calimeri *et al.*, since 2003]. The overall architecture of the system is shown in Figure 2.

The DLP^T system work-flow can be described as follows.

A DLP^T program is sent to a DLP^T pre-parser, which performs syntactic checks (included non-recursivity checks), and builds an internal representation of the DLP^T program. The DLP^T Inflater performs the *Explode* Algorithm and produces an equivalent DLV program P'; P' is piped towards the DLV system. The models $M(P')$ of P', computed by DLV, are then converted in a readable format through the Post-parser module; the Post-parser filters out from $M(P')$ informations about internally generated predicates and rules.

The system introduces also some useful features in order to ease programming. For instance, the possibility to define some predicates as 'global', just specifying them in the template definition.

$$\textbf{\#template } n_D[f_1(b_1), \cdots, f_n(b_n)](b_{n+1})$$
$$\textbf{GLOBAL } g_1, \cdots, g_m$$

where $g_1, \cdots, g_m$ is a list of predicate symbols defined as global. This introduces the notion of *scope*. The notion is similar to traditional imperative languages, such as C++, where it is possible to mask global variables. Intuitively, the meaning of the local predicates results from the rules defined within the template body, while the meaning of the global predicates results from the rules belonging to the general program. We refer to function scope in the former case, and program scope in the latter.

Example 11 In this template definition, `node` is a global predicate, while `coloring` is local, and `arc` is an argument.

```
#template coloring[arc(2)](2) GLOBAL node
{
coloring(Country, red) v
  coloring(Country, green) v
  coloring(Country, blue) :- node(Country).
:- arc(Country1, Country2),
  coloring(Country1, CommonColor),
  coloring(Country2, CommonColor).
}
```

7. Conclusions

In this paper we have addressed some lacks of DLP, namely code reusability and modularity. We have presented the DLP^T language, an extension of DLP allowing to define template predicates.

The proposed language is very promising; the future work will have as objectives:

- introducing a clearer model theoretic semantic and prove its equivalence with the current operational semantics;
- generalizing template semantics in order to allow safe and meaningful forms of recursion between template definitions;

– introducing new forms of template atoms in order to improve reusability of the same template definition in different contexts;
– prove the formal equivalence of DLT sub-programs with semantics for aggregate constructs such as in [Calimeri *et al.*, 2005];
– extending the template definition language using standard languages such as `C++`, such as in [Eiter *et al.*, 2005];
– consider program equivalence results [Eiter *et al.*, 2004] in order to optimize the size of unfolded programs.

The DLP^T system prototype is available at `http://dlt.gibbi.com`.

Acknowledgements

This work has been supported by the European Commission under the project INFOMIX (IST-2001-3357), and by FWF (Austrian Science Fund) under the project "Answer Set Programming for the Semantic Web" (P17212-N04).

We thank Adriana Pietramala and Maria Carmela Santoro for their former work on this project.
We would also like to thank the reviewers for their fruitful comments.

References

[Anger *et al.*, 2001] Christian Anger, Kathrin Konczak, and Thomas Linke. `NoMoRe`: A System for Non-Monotonic Reasoning. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 406–410. Springer Verlag, September 2001.

[Apt and Bol, 1994] K. Apt and N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[Aravindan *et al.*, 1997] Chandrabose Aravindan, Jürgen Dix, and Ilkka Niemelä. DisLoP: A Research Project on Disjunctive Logic Programming. *AI Communications – The European Journal on Artificial Intelligence*, 10(3/4):151–165, 1997.

[Babovich, since 2002] Yulia Babovich. Cmodels homepage, since 2002. http://www.cs.utexas.edu/users/tag/cmodels.html.

[Bell *et al.*, 1994] Colin Bell, Anil Nerode, Raymond T. Ng, and V.S. Subrahmanian. Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases. *Journal of the ACM*, 41:1178–1215, 1994.

[Buccafurri *et al.*, 2000] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[Bugliesi *et al.*, 1994] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.

[Cadoli and Schaerf, 2001] Marco Cadoli and Andrea Schaerf. Compiling Problem Specifications into SAT. In *ESOP*, pages 387–401, 2001.

[Cadoli *et al.*, 1999] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: An Executable Specification Language for Solving all Problems in NP. In Gopal Gupta, editor, *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science, pages 16–30. Springer, 1999.

[Calimeri *et al.*, 2005] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Simona Perri. Declarative and Computational Properties of Logic Programs with Aggregates. In *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 406–411, August 2005.

[Calimeri *et al.*, since 2003] Francesco Calimeri, Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro. The DLP^T homepage, since 2003. http://dlpt.gibbi.com/.

[Chen and Warren, 1996] Weidong Chen and David Scott Warren. Computation of Stable Models and Its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.

[Cholewiński *et al.*, 1996] Paweł Cholewiński, V. Wiktor Marek, and Mirosław Truszczyński. Default Reasoning System DeReS. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR '96)*, pages 518–528, Cambridge, Massachusetts, USA, 1996. Morgan Kaufmann Publishers.

[Cholewiński *et al.*, 1999] Paweł Cholewiński, Victor W. Marek, Artur Mikitiuk, and Mirosław Truszczyński. Computing with Default Logic. *Artificial Intelligence*, 112(2–3):105–147, 1999.

[Dantsin *et al.*, 2001] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[Dell'Armi *et al.*, 2003] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, pages 847–852, Acapulco, Mexico, August 2003. Morgan Kaufmann Publishers.

[Dix, 1995] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In *Logic, Action and Information. Proceedings of the Konstanz Colloquium in Logic and Information (LogIn'92)*, pages 241–329. DeGruyter, 1995.

[East and Truszczyński, 2000] Deborah East and Mirosław Truszczyński. dcs: An Implementation of DATALOG with Constraints. In Chitta Baral and Mirosław Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.

[East and Truszczyński, 2001] D. East and M. Truszczyński. Propositional Satisfiability in Answer-set Programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, pages 138–153. Springer Verlag, LNAI 2174, 2001.

[Egly *et al.*, 2000] Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00), July 30 – August 3, 2000, Austin, Texas USA*, pages 417–422. AAAI Press / MIT Press, 2000.

[Eiter *et al.*, 1997a] Thomas Eiter, Georg Gottlob, and Nicola Leone. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.

[Eiter *et al.*, 1997b] Thomas Eiter, Georg Gottlob, and Helmuth Veith. Modular Logic Programming and Generalized Quantifiers. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-97)*, number 1265 in LNCS, pages 290–309. Springer, 1997.

[Eiter *et al.*, 2000] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[Eiter *et al.*, 2004] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Simplifying logic programs under uniform and strong equivalence. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, number 2923 in Lecture Notes in AI (LNAI), pages 87–99, Fort Lauderdale, Florida, USA, January 2004. Springer.

[Eiter *et al.*, 2005] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *International Joint Conference on Artificial Intelligence (IJCAI) 2005*, pages 90–96, Edinburgh, UK, August 2005.

[Faber and Pfeifer, since 1996] Wolfgang Faber and Gerald Pfeifer. `DLV` homepage, since 1996. http://www.dlvsystem.com/.

[Faber *et al.*, 1999] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.

[Faber *et al.*, 2001] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.

[Ferraris and Lifschitz, 2005] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1–2):45–74, 2005.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[Ianni *et al.*, 2003] Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro. Answer Set Programming with Templates. In Marina de Vos and Alessandro Provetti, editors, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 239–252, Messina, Italy, September 2003. Online at http://CEUR-WS.org/Vol-78/.

[ICONS, since 2001] ICONS homepage, since 2001. http://www.icons.rodan.pl/.

[Janhunen and Niemelä, 2004] Tomi Janhunen and Ilkka Niemelä. Gnt - a solver for disjunctive logic programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, number 2923 in Lecture Notes in AI (LNAI), pages 331–335, Fort Lauderdale, Florida, USA, January 2004. Springer.

[Kuper, 1990] G. M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41(1):44–64, 1990.

[Laenens *et al.*, 1990] Els Laenens, Domenico Saccà, and Dirk Vermeir. Extending Logic Programming. In *SIGMOD Conference*, pages 184–193, 1990.

[Leone and Rullo, 1993] Nicola Leone and Pasquale Rullo. Ordered Logic Programming with Sets. *Journal of Logic and Computation*, 3(6), December 1993.

[Leone *et al.*, 2005a] Nicola Leone, Georg Gottlob, Riccardo Rosati, Thomas Eiter, Wolfgang Faber, Michael Fink, Gianluigi Greco, Giovambattista Ianni, Edyta Kałka, Domenico Lembo, Maurizio Lenzerini, Vincenzino Lio, Bartosz Nowicki, Marco Ruzzi, Witold Staniszkis, and Giorgio Terracina. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 915–917, Baltimore, Maryland, USA, June 2005. ACM Press.

[Leone *et al.*, 2005b] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions* *on Computational Logic*, 2005. To appear. Available via http://www.arxiv.org/ps/cs.AI/0211004.

[Lierler and Maratea, 2004] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, LNCS, pages 346–350. Springer, January 2004.

[Lierler, 2005] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.

[Lin and Zhao, 2004] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[Marek and Remmel, 2002] Victor W. Marek and Jeffrey B. Remmel. On Logic Programs with Cardinality Constraints. In Salem Benferhat and Enrico Giunchiglia, editors, *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, pages 219–228, Toulouse, France, April 2002.

[McCain and Turner, 1998] Norman McCain and Hudson Turner. Satisfiability Planning with Causal Theories. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 212–223. Morgan Kaufmann Publishers, 1998.

[Niemelä *et al.*, 1999] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable Model Semantics of Weight Constraint Rules. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 107–116, El Paso, Texas, USA, December 1999. Springer Verlag.

[Niemelä *et al.*, 2000] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A System for Answer Set Programming. In Chitta Baral and Mirosław Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.

[Niemelä, 1999] Ilkka Niemelä. Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[Nogueira *et al.*, 2001] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog Decision Support System for the Space Shuttle. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages, Third International Symposium (PADL 2001)*, number 1990 in Lecture Notes in Computer Science, pages 169–183. Springer, 2001.

[Rao *et al.*, 1997] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[Ross and Sagiv, 1997] K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *Journal of Computer and System Sciences*, 54(1):79–97, February 1997.

[Seipel and Thöne, 1994] Dietmar Seipel and Helmut Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. In Antoni Olivé, editor, *Proceedings International Workshop on the Deductive Approach to Information Systems and Databases (DAISD'94)*, pages 325–343. Universitat Politecnica de Catalunya (UPC), 1994.

[Simons, 1999] Patrik Simons. Extending the Stable Model Semantics with More Expressive Rules. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 305–316, El Paso, Texas, USA, December 1999. Springer Verlag.

[Simons, 2000] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland, 2000.

[Subrahmanian *et al.*, 1995] V.S. Subrahmanian, Dana Nau, and Carlo Vago. WFS + Branch and Bound = Stable Models. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):362–377, June 1995.