DLGP: An extended Datalog Syntax for Existential Rules and Datalog± *Version 2.1*

GraphIK Team[*], LIRMM / Inria, Montpellier

February 9, 2017

Abstract

This document specifies the version 2.1 of *dlgp*, a textual format for the existential rule / Datalog± framework. This format is meant to be an exchange format at once human-friendly, concise and easy to parse. It can be seen as an extension of the commonly used format for plain Datalog. It is called "dlgp" for "Datalog Plus". A file may contain four kinds of knowledge elements: facts, existential rules, negative constraints and conjunctive queries.

1 Change logs

1.1 v2.1

- fixes a bug when parsing a variable matching the following pattern E[0-9]+;
- allows to write query with empty list of answer variables like "`?()`:- p(X)." in addition to "`?`:- p(X).";
- allows empty query "`?`:- ." and so rule like "`p(a)`:- ." is also allowed;
- relaxes the order constraint on elements of the header part (@base, @prefix, @top and @una). The annotations @top and @base must still be unique.

1.2 v2.0

- The main improvement in this version is the introduction of Web notions (IRI and literal, following Turtle format) to ensure compatibility with Semantic Web languages;

[*]Jean-François Baget, Alain Gutierrez, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, Clément Sipieter.

- allows spaces in label;
- The only incompatiblity with Version 1.0 is the impossibility to use a quoted string to denote a predicate. Now, predicates have to be <l-ident>, <IRIREF> or <PrefixedName>. Constants have been extended: they may also be <IRIREF>, <PrefixedName> or any <literal> (not only integer, float et quoted string).

2 Introduction

The dlgp format encodes existential rules (and other constructs that can be seen as special kinds of existential rules: facts, negative constraints and conjunctive queries). A basic logical notion is that of an *atom*, which is composed of a *predicate* (or relation name) and arguments called *terms*. Terms can be variables or constants. Predicates can be of any arity greater or equal to one.

In their simplest form, predicates and terms are encoded by string identifiers built with *letters* from the Latin aphabet, *digits* and the *underscore* character (_). As usually in Datalog, variables begin with an uppercase letter, while constants and predicates begin with a lowercase letter. To make the dlgp format compatible with data from the Semantic Web, more elaborate kinds of identifiers are mandatory. The version 2.x introduces the notions of IRI and literal, according to Turtle format (`http://www.w3.org/TR/turtle/`). Specifically:

- a predicate or a constant can be denoted by an *IRI* in Turtle format, which can be written in three forms: a relative IRI, an absolute IRI or a prefixed name. Relative and absolute IRIs are strings of the form `<iri>` (see the symbol IRIREF in Turtle grammar). A prefixed name is a string prefixed by a namespace (see the symbol PrefixedName in Turtle grammar); moreover, the classical Datalog format (a string beginning with a lowercase letter) is allowed and understood as a relative IRI;
- a constant can also be a literal (see the symbol literal in Turtle grammar).

For instance, here are different ways of writing a predicate:

- `pred` (classical logic programming notation);
- `<pred>` (a relative IRI, whose base is specified elsewhere);
- `<Pred>` (a relative IRI as well, starting with an uppercase letter);
- `<http://exemple.org/pred>` (an absolute IRI);

- `ex:pred` (prefixed name); in this case the IRI associated with the prefix `ex` has to be introduced by a specific annotation in the file header, e.g., `@prefix ex: <http://exemple.org/>`.

All these forms are usable to encode a constant. Moreover, constants can be described by *literals*, e.g., `-5.1`, `true`, `"constant"`, or any other Turtle literal. Note that the tokens `true` and `false` are interpreted as Boolean literals and not as IRIs.

The special predicate $=$ encodes equality between terms (e.g., $X = Y$).

Sets of such atoms are logically interpreted as *conjunctions* of atoms. The following four kinds of knowledge elements are built upon sets of atoms:

- *facts* : a fact is a set of atoms interpreted as an existentially closed conjunction, i.e., of the form $\exists X \mathcal{F}[X]$, where X denote the set of variables occurring in $\mathcal{F}$; a fact can also be seen as an existential rule with an empty body;

- *(pure) existential rules* : an existential rule is an ordered pair of atom sets $(\mathcal{B}, \mathcal{H})$ interpreted as $\forall X(\exists Y \mathcal{B}[X,Y] \rightarrow \exists Z \mathcal{H}[X,Z])$, or, equivalently, as $\forall X \forall Y(\mathcal{B}[X,Y] \rightarrow \exists Z \mathcal{H}[X,Z])$, where X, Y and Z denote sets of variables ($\mathcal{B}$ is built on X and Y, while $\mathcal{H}$ is built on X and Z, hence X denotes the set of variables shared by $\mathcal{B}$ and $\mathcal{H}$);

- *negative constraints* : a negative constraint is a set of atoms interpreted as the negation of the corresponding fact, i.e., $\neg(\exists X \mathcal{C}[X])$; equivalently, it can be seen as a rule with a head restricted to the always-false symbol $\bot$: $\forall X(\mathcal{C}[X] \rightarrow \bot)$;

- *conjunctive queries* : a conjunctive query is a set of atoms provided with special variables (called the answer variables), interpreted as an existentially quantified conjunction where answer variables are kept free. It can also be seen as a rule of the form $\forall X \forall Y(\mathcal{B}[X,Y] \rightarrow ans(X))$, where ans is a special predicate and X is the set of answer variables.

A dlgp document is any sequence of such elements. The file name has extension `.dlp` or `.dlgp`. Characters are assumed to be encoded in UTF-8. Analysis directives are introduced by the symbol @ and comments by the symbol %.

3 Syntax of the dlgp format

In the following the syntax is specified by a grammar in BNF style: non-terminal symbols are enclosed in angle brackets <>, terminal symbols are in bold font, the | symbol indicates a choice, parts enclosed in square brackets

([]) are optional, *choice*$_1$..*choice*$_n$ indicates a choice within an interval; parts enclosed in braces can be repeated from 0 to n times ({*repeated-pattern*}*) or from 1 to n times ({*repeated-pattern-at-least-once*}+).

3.1 Comments

Comments are introduced by the symbol % outside Turtle tokens (i.e., outside <IRIREF> and <literal>, which may themselves contain the symbol %). A comment ends at the end of the same line or at the end of the file. Moreover, comments introduced by %% can be interpreted in a specific way. Our parser generates an event when such a specific comment is read, which can be exploited by event listeners.

3.2 Parsing Information

Please refer to Turtle syntax for building an absolute IRI from a relative IRI (and @base directives) or from a prefixed name (and @prefix directives). Note that the directive base may occur at most once. Similarly, it is not possible to successively assign several IRIREF to the same prefix (@prefix directive). These two directives may occur only in the head of the file.

A <l-ident> token is seen as a relative IRI, and the corresponding absolute IRI is obtained by adding the IRI of the @base directive in front of the <l-ident> token.

3.3 Elements used to define tokens

<uppercase-letter> ::=	**A**..**Z**
<lowercase-letter> ::=	**a**..**z**
<digit> ::=	**0**..**9**
<underscore> ::=	_
<letter> ::=	<uppercase-letter> | <lowercase-letter>
<simple-char> ::=	<letter> | <digit> | <underscore>
<PN_CHARS>	*see turtle grammar*
<space> ::=	#x20 /* *#x20 = space character* */

3.4 Tokens

<u-ident> ::=	<uppercase-letter> {<simple-char>}*
<l-ident> ::=	<lowercase-letter> {<simple-char>}*
<label> ::=	{<PN_CHARS> | <space>}*

3.5 Global Grammar

```
<document> ::=                  <header> <body>
<header> ::=                    { <base> | <prefix> | <top> | <una> }*
<base> ::=                      @base <IRIREF>
<prefix> ::=                    @prefix <PNAME_NS> <IRIREF>
<top> ::=                       @top <l-ident> |
                                @top <IRIREF>
<una> ::=                       @una
<body> ::=                      {<statement>}* |
                                {<section>}*
<section> ::=                   @facts {<fact>}* |
                                @rules {<rule>}* |
                                @constraints {<constraint>}* |
                                @queries {<query>}*
<statement> ::=                 <fact> | <rule> | <constraint> | <query>
<fact> ::=                      [ [<label>] ] <not-empty-conjunction>.
<rule> ::=                      [ [<label>] ] <not-empty-conjunction> :- <conjunction>.
<constraint> ::=                [ [<label>] ] ! :- <not-empty-conjunction>.
<query> ::=                     [ [<label>] ] ? [(<term-list>)] :- <conjunction>.
<conjunction> ::=               [<not-empty-conjunction>]
<not-empty-conjunction> ::=     <atom> {, <atom>}*
<atom> ::=                      <std-atom> | <equality>
<equality> ::=                  <term> = <term>
<std-atom> ::=                  <predicate>(<not-empty-term-list>)
<term-list> ::=                 [<not-empty-term-list>]
<not-empty-term-list> ::=       <term> {, <term>}*
<term> ::=                      <variable> | <constant>
<predicate> ::=                 <l-ident> | <IRIREF> | <PrefixedName>
<variable> ::=                  <u-ident>
<constant> ::=                  <l-ident> | <IRIREF> | <PrefixedName> | <literal>
```

The symbol @ is used to introduce several kinds of annotations:

- In the header:
  - `@base` is a Turtle directive which defines a base to complement relative IRIs. This annotation is not mandatory even if relative IRIs are used in the document; a base IRI is then defined by the context of the application. This annotation must be unique.
  - `@prefix` is a Turtle directive that defines a pair *prefix / namespace identifier* used to build an IRI from a prefixed name (the prefix is a string terminated by a colon and the namespace identifier is an <IRIREF>). The IRI is obtained by replacing the prefix in the prefixed name by the IRI identifying the namespace

associated with that prefix. All the prefixes occurring in the document must be defined by a `@prefix` annotation. All the prefixes occurring in `@prefix` annotations must be pairwise distinct.

  - `@top` defines a predicate whose semantics is $\top$ (i.e., the knowledge base satisfies the axiom $\forall x \top(x)$, where $\top$ is replaced by this predicate). This annotation must be unique.
  - `@una` states that this knowledge base makes the Unique Name Assumption, i.e., all constants are assumed to refer to distinct objects.

- In the body:
  - the directives `@facts`, `@rules`, `@constraints`, `@queries` can be used to declare the nature of the elements that follow (respectively: facts, rules, constraints and queries). Such a directive defines a section that spans to the next annotation or to the end of the file. The interest of this kind of directive is to inform sooner the parser of the nature of the next statements. For instance, without these directives, it is not possible to distinguish between a fact and a rule before the sequence of atoms composing the fact or the rule head has been completely analyzed. However, it may be interesting to know that a fact is coming to process it more efficiently, specially if large facts can be encountered. However, such a directive is only an information to the parser, which can choose to ignore it.
  - each element can be optionally preceded by a *label* enclosed in square brackets. This name is any string composed of <`PN_CHARS`> in Turtle format as well as the space character.

To encode other kinds of information about the knowledge base, it is recommended to use specific comments introduced by %%.

Note on the scope of variable identifiers. While the scope of a constant or a predicate identifier is the whole document, the scope of a variable is local to a <statement>. Thus two different facts, rules or constraints actually do not share any variable (more precisely, variables with the same name in different statements are each bound by their own quantifier).
Example:
`p(X,a), q(X,Y).`
`q(X,b).`
is logically interpreted as $\exists X \exists Y (p(X,a) \wedge q(X,Y)) \wedge \exists X q(X,b)$
while:
`p(X,a), q(X,Y), q(X,b).`
is logically interpreted as $\exists X \exists Y (p(X,a) \wedge q(X,Y) \wedge q(X,b))$.

4 Examples

4.1 A simple example using annotations

```
@facts
[f1] p(a), relatedTo(a,b), q(b).
[f2] p(X), t(X,a,b), s(a,z).
t(X,a,b), relatedTo(Y,z).

@constraints
[c1] ! :- relatedTo(X,X).
[constraint_2] ! :- X=Y, t(X,Y,b).
! :- p(X), q(X).

@rules
[r1] relatedTo(X,Y) :- p(X), t(X,Z).
s(X,Y), s(Y,Z) :- q(X),t(X,Z).
[rA 1] p(X) :- q(X).
Y=Z :- t(X,Y),t(X,Z).
s(a) :- .
s(Z) :- a=b, X=Y, X=a, p(X,Y).

@queries
[q1] ? (X) :- p(X), relatedTo(X,Z), t(a,Z).
[Query2] ? (X,Y) :- relatedTo(X,X), Y=a.
? :- p(X).
?() :- .
```

4.2 An example which illustrates the use of IRIs and literals

The three first facts have the same interpretation.

```
@base <http://www.example.org/>
@prefix ex: <http://www.example.org/>
@prefix inria-team: <https://team.inria.fr/>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>

@facts

% use of @base
[f 1] <Pred>(1.5).

% use of @prefix
[f 2] ex:Pred("1.5"^^xsd:decimal).

% absolute IRIs
[f 3] <http://www.example.org/Pred>
      ("1.5"^^<http://www.w3.org/2001/XMLSchema#decimal>).

% use of @base for the predicate and @prefix for the argument
[f 4] team(inria-team:graphik).
```

4.3 A syntactically correct but not human-friendly file

```
[f1] p(a), relatedTo(a,b), q(b). [f2] p(X), t(X,a,b), s(a,z).
[c1] !:-relatedTo(X
% this is a comment
,X).
[q1]?(X) :- p(X), relatedTo(X,Z), t(a,Z).
t(X,a,b).
[r1] relatedTo(X,Y) :- p(X), t(X,Z).
[constraint_2] ! :- X=Y, t(X,Y,b).
s(X,Y), s(Y,Z) :- % This is another comment
q(X),t(X,Z).
[rA_1] p(X)
:-
q(X)
. Y=Z :- t(X,Y),t(X,Z).
[Query2]
? (X,Y) :- relatedTo(X,X), Y=a.
s(Z) :- a=b, X=Y, X=a, p(X,Y).
!:- p(X), q(X).
 relatedTo(Y,z).?    :- p(X).
```