Creation of NLS-GFT-OPI Corpus

This document describes:

1. the structure of a NLS-GFT-OP corpus and its ingredients;
2. the process of the creation of a NLS-GFT-OP corpus.

There are two prerequisites for the corpus creation: an ontology and GF grammars. We suppose that these two are given apriori and we want to create the corpus. The aim of such a corpus is to store examples of correspondence between:

1. ontology pattern instances (OPI);
2. natural language sentences (NLS);
3. GF trees (GFT);

What is a pattern (*ontology pattern*)? **What is** a pattern instance (*ontology pattern instance*)? Ontology pattern is a variant of *small directed graph*, whose nodes are ontology classes. The directed links between the nodes are labeled. A directed link from node X to node Y is labeled with an ontology property whose domain is a superclass of X and whose range is a superclass of Y. For example let Person be the class (the node) X and let also Person be the class (the node) Y. Let IsInLoveWith be the label of the directed link from X to Y. We have the small graph Person-IsInLoveWith-Person. This small graph does not represent the pattern by itself. To receive a pattern from the small graph we attach additional information(a flag) to every node. This flag differentiates the nodes into two types:

1. nodes that represent a concrete instance of the class;
2. nodes that represent any instance of the class.

Therefore the graph Person-IsInLoveWith-Person corresponds to four different patterns:

1. Person(concrete)-IsInLoveWith-Person(concrete)
2. Person(concrete)-IsInLoveWith-Person(any)
3. Person(any)-IsInLoveWith-Person(concrete)
4. Person(any)-IsInLoveWith-Person(any)

We use 'concrete' and 'any' as the two values of the additional flag attached to a node. A small graph with K nodes has 2^K corresponding patterns. We introduce also the notion *pattern instance.* Every pattern may have many pattern instances. A pattern is transformed into a pattern instance by replacing every 'concrete' node with a concrete instance of the represented class. For example consider that Mary, John, Mr Smith and Mrs Smith are instances of the class Person in the given ontology. Then:

1. the pattern Person(concrete)-IsInLoveWith-Person(concrete) may be transformed into the pattern instances Mary-IsInLoveWith-John, Mary-IsInLoveWith-Mary, Mrs Smith-IsInLoveWith-Mr Smith etc;
2. the pattern Person(concrete)-IsInLoveWith-Person(any) may be transformed into the pattern instances Mary-IsInLoveWith-Person(any), Mr Smith-IsInLoveWith-Person(any) etc;
3. the pattern Person(any)-IsInLoveWith-Person(concrete) may be transformed into the pattern instances Person(any)-IsInLoveWith-Mary etc.
4. the pattern Person(any)-IsInLoveWith-Person(any) has no 'concrete' nodes and represents a pattern instance by itself.

What is the semantic of a pattern instance? The semantic of a given pattern instance naturally inherits the semantic of the given ontology. In (description) logics a pattern instance with K nodes is a K-ary term with free typed variables corresponding to the 'any' nodes and constants corresponding to the 'concrete' nodes. The semantic of a pattern instance is the semantic of the K-ary term it represents. More intuitively: pattern instance without 'any' nodes represents a relation of its concrete instances. Pattern instance with 'any' nodes represents a relation of its concrete instances and the sets of all possible instances corresponding to the 'any' nodes.

What is the interpretation of a pattern instance? The interpretation of a pattern instance depends on the application. We want to use the corpus for the following two applications at least:

1. Translation of a restricted natural language into a formal query language (like SPARQL and SERQL)
2. Generation in natural language of a textual descriptions of entities stored in the ontology.

Consider the first case - translation of natural language to SPARQL.
1. Every pattern instance with 'any' node(s) can be naturally interpreted as a (SPARQL) query to the given ontology. The transformation of a given pattern instance into a SPARQL query is straightforward. For example Person(any)-IsInLoveWith-Mary is syntactic sugar for:

```
select distinct ?person where { ?s rdf:type xxx:Person .
                                ?s rdfs:label "Mary" .
                                ?person rdf:type xxx:Person .
                                ?person yyy:IsInLoveWith ?s. }
```

where xxx and yyy are suitable namespaces. So we interpret every pattern instance with M > 0 'any' nodes as a SPARQL query with M variables.

2. Every pattern instance without 'any' nodes, i.e. pattern instance that contains only 'concrete' nodes, can be also interpreted as a query. For example the pattern instance Mary, whose pattern is Person(concrete), can be interpreted as the query:

```
select distinct ?person where { ?person rdf:type xxx:Person .
                                ?person rdfs:label "Mary" . }
```

It will extract all persons labeled Mary. We interpret every pattern instance with no 'any' nodes and L 'concrete' nodes as a SPARQL query with L variables.

For simplicity, the order of the variables may be fixed apriori for every pattern instance, although when I ask 'all persons and their job positions' I expect to see a table of two columns: the first one to contain persons and the second one – the corresponding job positions. However when I ask 'all job positions and their holders' I expect the first column to contain job positions and the second column – the corresponding persons.

In the second case (generation in natural language of a textual descriptions of entities from the ontology) we shall not use pattern instances with 'any' nodes. We shall use only pattern instances that contain only 'concrete' nodes.

What is a *natural language sentence*? By natural language sentence we mean a sentence whose meaning somehow corresponds to the semantic of a pattern instance. For example natural language sentences for the pattern instance Person(any)-IsInLoveWith-Mary may be:

1. Who is in love with Mary?
2. All persons being in love with Mary.
3. Person who is in love with Mary.

And so on.
Natural language sentences for the pattern instance Mary-IsInLoveWith-Mary may be:

1. Mary is in love with herself.
2. Mary loves herself.

And so on.

What is a GF tree? It is a tree representing the result of a parsing a given natural language sentence with the GF grammars. The input of GF consists of GF grammars and a sentence. The output of GF is GF tree. The GF tree represents the syntactic structure of the input language.

GF trees are generated by GF automatically from the natural language sentences, so GF trees may not be stored explicitly in the corpus. Only the correspondence between sentences and ontology pattern instances is to be kept explicitly in the corpus.

The process of corpus creation

1. Select manually a small directed graph G from the ontology.
2. Transform manually G into an interesting pattern P.
3. Generate a pattern instance I of P.
4. Generate natural language sentence S of I. Store in the corpus the triple (S,I,P).
5. If you have another natural language sentence of I, then go to 4 else go to 6.
6. If you know another interesting pattern instance of P, then go to 3 else go to 7.
7. If you know another interesting pattern of G, then go to 2 else go to 8.
8. If you know another interesting small directed graph, then go to G else go to 9.
9. End.

As one can see below storing the triple (S,I,P) is very simple if we know I and P: it means just to add a sentence in a txt file.

Structure of the corpus

For every small graph G there is a separate directory [G]. In [G] there is a file named 'graph.nt'. This file describes in ntriple format the graph G. For every pattern P of G there is a subdirectory [G/P] of [G]. For every pattern instance I of P there is a file [I].txt. This file is separated into sections. There may be the following three sections:

1. [pattern instance];
2. [query variables];
3. [sentences].

In the section [pattern instance] there is a description of the pattern instance in ntriple format. The name of each 'any' node starts with 'any'. The name of each 'concrete' node does not start with 'any'.

The section [query variables] is optional. It keeps the names of the 'any' nodes, which will be selected variables in the SPARQL query generated from the pattern instance. The order of the selected variables in the SPARQL query will be the order of the names in the section [query variables].

The section [sentences] contains the corresponding natural language sentences. There are two types of sentences: simple and complex. Simple sentences are marked with {simple}. Complex sentences are marked with {complex}. A sentence is considered {simple} if

1. it contains the names of all concrete nodes and
2. we can directly replace the names of the concrete nodes with other names.

For example let us consider
graph.nt:

```
<...JobPosition> <...holder> <...Person> .
<...JobPosition> <...hasTitle> <...JobTitle> .
<...JobPosition> <...withinOrganization> <...Organization> .
```

[pattern instance]

```
<anyJobPosition> <...holder> <anyPerson> .
<anyJobPosition> <...hasTitle> <president> .
<anyJobPosition> <...withinOrganization> <Microsoft> .
```

[query variables]
anyPerson

[sentences]
{simple} Who is the president of Microsoft?

The sentence is marked {simple}, because we can directly replace 'president' with each other job titles: CEO, CFO etc. and we can directly replace 'Microsoft' with each other organization: Ontotext, Google etc.

However with same graph we may have
[pattern instance]

```
<anyJobPosition1> <...holder> <Larry Page> .
<anyJobPosition1> <...hasTitle> <president> .
<anyJobPosition1> <...withinOrganization> <Google> .
<anyJobPosition2> <...holder> <Sergey Brin> .
<anyJobPosition2> <...hasTitle> <president> .
<anyJobPosition2> <...withinOrganization> <Google> .
```

[sentences]
{complex} Larry Page and Sergey Brin are presidents of Google.

The sentence is {complex}, because the concrete name 'president' is not used.