STTL: A SPARQL-based Transformation Language for RDF

Olivier Corby, Catherine Faron Zucker

STTL: A SPARQL-based Transformation Language for RDF

Olivier Corby[1] and Catherine Faron-Zucker[2]
[1]*Inria Sophia Antipolis Méditerranée, France*
[2]*Univ. Nice Sophia Antipolis, I3S, France*
olivier.corby@inria.fr, faron@unice.fr

Keywords: RDF Transformation Language, SPARQL, STTL

Abstract: The general research question addressed in this paper is *How to transform RDF into other languages*. This is of prime interest to present data selected and extracted from the Web of data in a format suitable for the user (e.g., HTML or CSV). Moreover, RDF can be viewed as a meta-model to represent on the Web of data other languages and models. The above research question then becomes *How to generate the concrete syntax of expressions of a given language from their RDF representation*. To answer these questions, we present SPARQL Template Transformation Language (STTL), a generic RDF transformation rule language, independent from the output language. We conceived it as a lightweight syntactic extension to SPARQL and we show how to compile it into standard SPARQL. We present a generic transformation rule engine implementing STTL and several RDF transformers we defined for various output languages, showing STTL's expressive power.

1 Introduction

The read-write Web is now providing us with a world-wide blackboard where a hybrid society of users and software agents exchange digital inscriptions. The RDF standard (Cyganiak et al., 2014) provides us with a general purpose graph-oriented data model recommended by the W3C to represent and interchange data on the Web. While the potential of a world-wide Semantic Web of linked data and linked data schemas is now widely recognized, the transformation and presentation of RDF data is still an open issue. Among the initiatives to answer this question there are extensive works for providing RDF with several and varied syntaxes: XML, N-Triples, Turtle, RDFa, TriG, N-Quads, JSON-LD, CSV-LD, etc. But this is still a partial view of the above problem.

Just like the structured Web has been provided with the XSLT transformation language to present XML data to the user into HTML pages or to transform XML data from one XML schema into another one or from an XML schema into any non XML specific text format, for XML data interchange between agents and therefore interoperability, the Web of data now requires a transformation language to present RDF data to users and transform RDF data from one RDF schema into another or transform data from its RDF "syntax" into another one. Indeed, a special case of RDF data holds a very special potential: RDF data encoding other formal languages. In computer science, formal languages have been used for instance to define programming languages, query languages, data models and formats, knowledge formalisms, inference rules, etc. Among them, in the early 2000's, XML has gained the status of a meta-language or syntax enabling to define so-called XML languages. In the same way, we are now assisting to the advent of RDF that will likely be more and more used as a syntax to represent other languages. For instance in the domain of the Semantic Web alone, this is the case of three W3C standards: OWL 2 (Peter F. Patel-Schneider and Motik, 2012) is provided with several syntaxes, among which the Functional syntax, the Manchester syntax used in several ontology editors and RDF/XML and RDF/Turtle; the Rule Interchange Format (RIF) (Sandro Hawke, 2012) is provided with several syntaxes among which a verbose XML syntax, two compact syntaxes for RIF-BLD and RIF-PRD and an RDF syntax; SPARQL Inference Notation (SPIN) is a W3C member submission (Knublauch, 2011) to represent SPARQL rules in RDF, to facilitate storage and maintenance. Many other languages can (and will) be "serialized" into RDF. For instance (Follenfant et al., 2012) is an attempt to represent SQL expressions in RDF.

As a result of this emerging trend to use RDF

as a "syntax" or a meta-language for other Web languages, just like XML ten years earlier, the transformation of RDF data into various output formats, various concrete syntaxes, becomes a major issue. Regarding the RDF/XML syntax of RDF, one could think that XSLT is a good candidate to declaratively express RDF transformations. However, writing of XSLT templates for RDF would be based on its XML syntax and not the graph structure and semantics of RDF model — the transformation rules would depend on the concrete XML syntax of RDF instead of its semantics—, which would make the writing of the transformation quite difficult. Moreover, the many potential serializations of any given RDF statement would make the writing XSLT templates for RDF even more complex.

In this paper we address the latter problem of transforming RDF data, i.e., generating the concrete syntax of expressions of a given language from their RDF representation. More generally, the research question addressed in this paper is *How to transform RDF data into other languages?* We answer two subquestions: (1) How to write declarative transformation rules from RDF to other languages? (2) How to make the approach generic, i.e the rule language independent from the output language?

We show how SPARQL (Harris and Seaborne, 2012) can be used as a generic transformation rule language for RDF, independent from the output languages. We define an RDF transformer as a set of transformation rules processed by a generic transformation rule engine. We present SPARQL Template Transformation Language (STTL), a lightweight syntactic extension to SPARQL enabling the writing of transformation rules.

In section 2 we present existing transformation languages for RDF. In section 3 we present STTL, a syntactic extension to SPARQL enabling the writing of RDF transformation rules. In section 4 we present the generic transformation rule engine we developed to implement STTL. In section 5 we show the expressive power of STTL through several RDF transformers we defined for various output languages.

2 Related Work

XSLT (Kay, 2007) is a pioneer rule-based transformation language for XML. An XSLT stylesheet is a set of transformation rules, called *templates*, which enables to transform any XML document conform to a given model, i.e., to which the templates apply. An XPath expression identifies the XML subtrees (their roots) for which a template applies, and the content of the template describes the transformation and its output. XSLT could be used to process and display RDF/XML data in any output format. For instance the following XSLT template could be used to transform RDF triples into an HTML table row.

```
<xsl:template
    match='rdf:Description[@rdf:about]'>
  <xsl:for-each select='./*'>
    <tr> <td>
    <xsl:value-of select='../@rdf:about'/>
    </td>
    <td> <xsl:value-of select='name()'/>
    </td>
    <td> <xsl:call-template name="value">
    <xsl:with-param name='v' select='.'/>
    </xsl:call-template>
    </td> </tr>
  </xsl:for-each>
</xsl:template>
```

However RDF/XML syntax is too versatile and less and less used and, most of all, writing XSLT templates for it would be very complex considering the many potential serializations of an RDF statement: the transformation rules would depend on the concrete XML syntax of RDF instead of its semantics.

GRDDL (Connolly, 2007) is a mechanism for extracting RDF data from XML documents. A GRDDL profile is associated to an XSLT stylesheet and can be specified in any XML document conform to the targeted model or *dialect* to order GRDDL agents to extract RDF data from it. GRDDL could then be used to extract RDF data from RDF data in RDF/XML syntax. However this W3C recommandation has never really been adopted and, like XSLT, this solution would rely on the many concrete XML syntaxes of RDF instead of its semantics.

OWL-PL (Brophy and Heflin, 2009) is an extension of XSLT for transforming RDF/OWL into XHTML. It provides an adaptation of XSLT processing of XML trees to RDF graphs. In particular, it matches properties of resources instead of XML nodes through XPath. OWL-PL is both tied to its RDF/XML input format, like XSLT. Xenon (Quan, 2005) is another ontology for specifying in RDF how RDF resources should be presented to the user. It reuses many of the key ideas of XSLT, among which templates, and defines a so-called RDF Stylesheet language. Xenon's two foundational concepts are lenses and views. Lenses specify which properties of an RDF resource are displayed and how these properties are ordered; views specify how they are displayed. Both OWL-PL and Xenon are tied to a specific display paradigm and an XHTML-like output format.

```
PersonLens a fresnel:Lens ;
  fresnel:classLensDomain foaf:Person ;
  fresnel:showProperties
    ( foaf:name foaf:mbox foaf:depiction ).
:nameFormat a fresnel:Format ;
  fresnel:label "Name" ;
  fresnel:propertyFormatDomain foaf:name .
```

Figure 1: A Fresnel RDF graph describing a presentation format for RDF data on persons

Fresnel (Bizer et al., 2005) is an RDF vocabulary for specifying in RDF which data contained in an RDF graph should be displayed and how. Fresnel's two foundational concepts are lenses and formats. Fresnel's formats generalize Xenon's views. Figure 1 presents a Fresnel RDF graph describing a presentation format for RDF data on persons: a lense specifies that for each person, her name, mbox and picture should be displayed and a format specifies how to display her name.

SPARQL is provided with a CONSTRUCT query form which enables to extract and *transform* RDF data into any other schema. A CONSTRUCT query returns an RDF graph specified by a graph template in the CONSTRUCT clause of the query and built by substituting the variables in the graph template with the solutions to the WHERE clause.

(Alkhateeb and Laborie, 2008) addresses the problem of generating XML from RDF data with an extended SPARQL query. A SPARQL query is given a template of XML document where variables are fed with the query results. The SPARQL CONSTRUCT clause is overloaded to refer to an XML template with reference to SPARQL query variables that are bound by a standard WHERE clause.

XSPARQL (Bischof et al., 2012) is a combination of SPARQL and XQuery (Robie et al., 2014) enabling to query both XML and RDF data and to transform data from one format into the other. XPARQL integrates within XQuery the SPARQL WHERE clause to facilitate the selection of RDF data to transform it into XML and, conversely, the SPARQL CONSTRUCT clause to facilitate the construction of RDF graphs from some extracted XML data. For instance the XSPARQL statements in FIgure 2 enable to select RDF data on persons and transform it into an XML tree describing relations between persons.

Finally, there are quite a wide range of RDF parsers and validators[1], some of which enable to transform RDF data from one serialization format to another. Among them, let us cite RDF Distiller[2] and

```
declare foaf="http://xmlns.com/foaf/0.1/";
<relations> {
  for $Person $Name from <relations.rdf>
  where {$Person foaf:name $Name}
  order by $Name
  return
    <person name = "{$Name}"> {
    for $FName
    where {$Person foaf:knows $Friend .
      $Friend foaf:name $FName}
    return <knows>{$FName}</knows> }
    </person> }
</relations>
```

Figure 2: XSPARQL statements to select RDF data on persons and transform it into an XML tree describing relations between persons

RDF Translator[3]. A review of these RDF-to-RDF converters can be found in (Stolz et al., 2013). Another famous example of specific-purpose RDF transformer is the RDF/XML parser in OWL API[4] (Horridge and Bechhofer, 2011) which enable to transform OWL 2 statements in RDF/XML into the Functional syntax of the language.

To sum up, the state-of-the-art solutions presented in this section to transform RDF data are all tied to either an RDF/XML input syntax or to a specific output format, or both — except Fresnel. But Fresnel focuses on the *presentation* of RDF data and does not handle the general problem of the *transformation* of RDF data. In the following, we present a *generic* approach for writing RDF transformers for any output language.

3 SPARQL Template Transformation Language

SPARQL Template Transformation Language (STTL) is a generic transformation rule language for RDF based on SPARQL. It relies on two extensions of SPARQL: an additional TEMPLATE query form to express transformation rules and extension functions to recursively call the processing of a template into another one. Section 3.1 summerizes the key features of SPARQL and sections 3.2 and 3.3 present the extensions of SPARQL in STTL. Section 3.4 presents STTL syntax; section 3.5 presents the compilation of STTL into standard SPARQL; and section 3.6 presents STTL semantics. Finally, section 3.7 compares STTL to XSLT.

[1] http://www.w3.org/2001/sw/wiki/Category:Tool

[2] http://rdf.greggkellogg.net/distiller

[3] http://rdf-translator.appspot.com/

[4] http://owlapi.sourceforge.net/

3.1 SPARQL

SPARQL is the query language for RDF recommended by W3C. It has a SQL-like syntax (SELECT FROM WHERE) and is a graph pattern matching language. A SPARQL query is a set of triple patterns that are RDF triples (in Turtle syntax) which may hold variables. A query may also have operators such as filter, conjunction, union, optional, minus, etc. An example of SPARQL query searching resources with `name` "Olivier" which are linked to other resources with a `knows` property is shown below:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE {
  ?x foaf:name "Olivier" ;
     foaf:knows ?y .
  FILTER (?x != ?y) }
```

SPARQL is also provided with a CONSTRUCT WHERE query form the result of which is a graph. The CONSTRUCT clause specifies a graph pattern with variables which are replaced by the values found in the solutions of the WHERE clause in order to create a graph. For instance, the following SPARQL query enables to construct the RDF graph of `foaf:knows` inverse relations between the resources of the original RDF graph.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT {
  ?y foaf:knows ?x }
WHERE {
  ?x foaf:knows ?y }
```

3.2 SPARQL Template Query Form

STTL relies on two extensions of SPARQL: an additional TEMPLATE query form and extension functions such as `st:apply-templates` to process a set of templates.

A TEMPLATE query is made of a standard WHERE clause and a TEMPLATE clause. The WHERE clause is the condition part of a rule, specifying the nodes in the RDF graph to be selected for the transformation. The TEMPLATE clause is the presentation part of the rule, specifying the output of the transformation for the RDF statements matching the condition.

For instance, let us consider the OWL 2 axiom stating that the class of men and the class of women are disjoint. Here is its expression in Functional syntax: `DisjointClasses(a:Man a:Woman)`; and here it is in Turtle: `a:Man owl:disjointWith a:Woman.` The following template enables to transform the above RDF statement into the corresponding statement in Functional syntax.

```
TEMPLATE {
  "DisjointClasses(" ?in " " ?c ")" }
WHERE { ?in owl:disjointWith ?c }
```

The WHERE clause matches the RDF statement and enables to select the subject and object of property `owl:disjointWith` and to bind them to variables `?in` and `?c`. The TEMPLATE clause specifies the result that must be generated using the solution sequence of the WHERE clause. Variables in the TEMPLATE clause are replaced by their value displayed in the Turtle syntax.

The value of a variable may be a blank node that represents another OWL statement, e.g., a Restriction. In this case, we would like to display not the blank node itself but the target OWL statement. This can be done using another template.

3.3 SPARQL Template Extension Functions

Let us now consider the OWL 2 axiom stating that the class of parents is equivalent to the class of individuals having a person as child. Here are its expressions in Functional syntax and in Turtle:

```
EquivalentClasses(
  a:Parent
  ObjectSomeValuesFrom(a:hasChild a:Person))

a:Parent a owl:Class ;
  owl:equivalentClass
    [ a owl:Restriction ;
      owl:onProperty a:hasChild ;
      owl:someValuesFrom a:Person ]
```

The following template enables to transform the above RDF statement into the corresponding Functional statement.

```
TEMPLATE {
 "EquivalentClasses("
 st:apply-templates(?in) " "
 st:apply-templates(?c) ")" }
WHERE { ?in owl:equivalentClass ?c . }
```

The value matching variable `?in` is `a:Parent` which is expected in the transformation output (the Functional syntax of the OWL 2 statement), while the value matching variable `?c` is a blank node whose property values are used to build the expected output.

This is defined in another template to be applied on this focus node. The `st:apply-templates` extension function[5] enables this recursive call of templates, where `st` is the prefix of STTL namespace: `<http://ns.inria.fr/sparql-template/>`. In

[5]Named in reference to XSLT `xsl:apply-templates`

other words, hierarchical processing is done using the `st:apply-templates` function in the TEMPLATE clause. It returns the result of the application of other templates to the focus nodes. The result is concatenated in the TEMPLATE clause. Hence, *nested templates* processing is performed by dynamic calls to `st:apply-templates`.

According to the definition of `PrimaryExpression` in SPARQL grammar, both SPARQL functions and extension functions can be used in TEMPLATE clauses. The following SPARQL extension functions have been defined to process a transformation:

- `st:apply-templates(term)` calls the transformer on a focus node `term` and executes one template;
- `st:call-template(name, term)` calls a template by its name on a focus node `term`;
- `st:apply-templates-with(uri, term)` calls the transformer specified by `uri` on a focus node `term` and executes one template;
- `st:call-template-with(uri, name, term)` calls a template by its name, on a focus node with a specified transformer;
- `st:apply-templates-all(term)` calls the transformer on a focus node `term` and executes all templates; it returns the concatenation of the results.

Some additional utilitary extension functions have been defined, among which:

- `st:turtle(term)` returns the Turtle form of an RDF term;
- `st:define()`, `st:process()` and `st:default()` enable to parameterize the transformation behaviour when used in the predefined templates presented in section 4.4.

3.4 Syntax

Figure 3 presents STTL's grammar. It is based on SPARQL 1.1's grammar[6]. In the definition of `Template`, `Prologue`, `DatasetClause`, `WhereClause`, `SolutionModifier` and `ValuesClause` are those defined in SPARQL grammar. In the definition of `TemplateClause`, `iri` is a template name. In the definition of `Term`, `PrimaryExpression` is that defined in SPARQL grammar and `Group` is syntactic sugar for SPARQL GROUP_CONCAT aggregate, enabling an easier

[6]http://www.w3.org/TR/sparql11-query/#sparqlGrammar

```
Template ::= Prologue TemplateClause
   DatasetClause* WhereClause
   SolutionModifier ValuesClause

TemplateClause ::=
  'TEMPLATE' (iri VarList ?) ?
  '{' Term* Separator? '}'

VarList ::= '(' Var+ ')'

Term ::= PrimaryExpression | Group

Group ::= 'GROUP' 'DISTINCT'? '{'
   PrimaryExpression* Separator? '}'

Separator ::= ';' 'separator' '=' String
```

Figure 3: STTL grammar

writing of aggregation with several arguments. `Separator` enables to define the separator of aggregates; by default, it is the space character for `Group` (see Section 3.5) and the newline character for `TemplateClause` (see Section 3.6).

3.5 Compilation into standard SPARQL

A template can be compiled into a standard SPARQL query of the SELECT form. The compilation keeps the WHERE clause, the solution modifiers and the VALUES clause of the template unchanged and the TEMPLATE clause is compiled into a SELECT clause. Here is the compilation scheme of a TEMPLATE clause into a SELECT clause:

```
(1) cp(TemplateClause(
      Term(t1), ... Term(tn), sep)) =
    SELECT (CONCAT(cp(t1), ... cp(tn))
    AS ?out)
(2) cp(Group(Term(t1), ... Term(tn), sep))
    =
    GROUP_CONCAT(CONCAT(cp(t1), ... cp(tn)),
    sep)
(3) cp(Var(v)) = st:process(v)
(4) cp(PrimaryExpression(if(e1, e2, e3))) =
    if(e1, cp(e2), cp(e3))
(5) cp(PrimaryExpression(e)) = e
```

Basically, a recursive function `cp` compiles a TEMPLATE clause by concatenating the compilation of its terms by a call to function CONCAT (1). A GROUP is syntactic sugar for GROUP_CONCAT aggregate (2); a variable `v` in a TEMPLATE clause is compiled into the `st:process(v)` function call (3); `if` function call arguments are compiled except the condition which is

left unchanged (4); other primary expressions are left unchanged (5).

For instance, by applying the above scheme, the following STTL expression:

```
TEMPLATE {
  "ObjectAllValuesFrom(" ?p " " ?c ")" }
WHERE {
  ?in a owl:Restriction ;
    owl:onProperty ?p ;
    owl:allValuesFrom ?c }
```

is compiled into the following SPARQL query:

```
SELECT
  (CONCAT("ObjectAllValuesFrom(",
   st:process(?p), " ",
   st:process(?c), ")") AS ?out)
WHERE {
  ?in a owl:Restriction ;
    owl:onProperty ?p ;
    owl:allValuesFrom ?c }
```

3.6 Evaluation Semantics

Since STTL can be compiled into standard SPARQL 1.1, the evaluation semantics of a template is that of SPARQL[7]. Let Ω the solution sequence resulting from the evaluation of the SPARQL query resulting itself from the compilation of a template. If the solution sequence is empty, the template fails. Otherwise, we define the result of the evaluation of a template as the result of the `Aggregation` operator of SPARQL Algebra[8] with the following arguments:

```
Aggregation((?out), GROUP_CONCAT,
    scalarvals, {1 -> Ω})
```

where `scalarvals` corresponds to the `sep` argument in the TEMPLATE clause. Matching the graph pattern in the WHERE clause of a template may return several solutions: Ω is a solution sequence. However, the result of the evaluation of a template is unique: the solutions in Ω are aggregated with a GROUP_CONCAT aggregate. The result of the template is the result of the GROUP_CONCAT.

3.7 Comparison of STTL and XSLT

STTL and XSLT are quite similar in their functionalities and expressiveness. The key difference between both languages is that XSLT operates on a XML (ordered) tree whereas STTL operates on an RDF (unordered) graph. Then, a XSLT template can match several patterns in the body whereas in a STTL template, there is one multiset of solutions resulting from the evaluation of one WHERE clause.

STTL inherits all SPARQL 1.1 statements, including Property Path and service which enables to perform Linked Data transformation on remote graphs.

4 Implementation

We implemented a SPARQL Template Transformation engine within the Corese Semantic Web Factory[9] (Corby et al., 2012; Corby and Faron-Zucker, 2010). Basically, it is called by the `st:apply-templates` extension function, or any other transformation functions introduced in section 3.3. Given an RDF graph with a focus node to be transformed and a list of templates, the transformation engine successively tries to apply them to the focus node until one of them succeeds. A template succeeds if the matching of the WHERE clause succeeds, i.e., returns a result.

4.1 Algorithm

Here is the core algorithm of the `st:apply-templates` function in pseudocode:

```
(1) Node st:apply-templates(Node node) {
(2)   for (Query q : getTemplates()) {
(3)     Mappings map = eval(q, IN, node);
(4)     Node res = map.getResult(OUT);
(5)     if (res != null) return res; }
(6)   return st:default(node); }
```

Templates are selected (2) and tried (3) one by one until one of them returns a result (4-5). In other words, a template is searched whose WHERE clause matches the RDF graph with the binding of the focus node to variable IN. If no template succeeds, the `st:default` function is applied to the node (6). Recursive calls to `st:apply-templates` implements the graph recursive traversal with successive focus nodes: `eval` (3) runs templates that recursively call the `st:apply-templates` function.

In addition to the above pseudocode, the transformer checks loops in case the RDF graph is a cyclic graph. It keeps track of the templates applied to nodes in order to avoid applying the same template on the

[7] http://www.w3.org/TR/sparql11-query/#sparqlAlgebraEval

[8] http://www.w3.org/TR/sparql11-query/#aggregateAlgebra

[9] http://wimmics.inria.fr/corese

same node twice. If no fresh template exists for a focus node, the transformer returns the value returned by a call to the `st:default` function.

A call to any other transformation functions introduced in section 3.3 triggers a similar algorithm.

4.2 Dynamic Variable Binding

When matching the WHERE clause of a template with the RDF graph, the SPARQL query evaluator is called with a binding of variable IN (`?in` in the WHERE clause) with the focus node to be transformed. When processing templates, the SPARQL interpreter must then be able to perfom dynamic binding to transmit the focus node. This dynamic value binding can be implemented in SPARQL with an extension function `st:getFocusNode()` that retrieves the focus node from the environment and a SPARQL BIND clause to bind it to the `?in` variable in the WHERE clause: `BIND(st:getFocusNode() AS ?in)`. The same scheme can be used for named templates with arguments.

4.3 Template Selection

By default, the transformation engine considers templates in order: given a focus node, in response to a call to the `st:apply-templates` function, it considers the first template that matches this node. Hence, the result of the transformation of the focus node is the result of this template. Named templates can be chosen to be processed by a call to the `st:call-template` function.

In some cases, it is worth writing *several* templates for a type of focus node, in particular when the node holds different graph patterns that should be transformed according to different presentation rules. Executing several templates on the focus node is done by calling the `st:apply-templates-all` function in the TEMPLATE clause. The result of the transformation is the concatenation of the results of the successful templates.

A transformer can be used to transform a whole RDF graph — without any distinguished root node in the graph. For this purpose, the `st:apply-templates- with` function can be called without focus node and the transformer must then determine it. By default, the first template that succeeds is the starting point of the transformer; or a `st:start` named template can be defined to be executed first (see Section 4.4).

4.4 Transformer Setting

Our implementation of STTL enables to simply set a special default template selection behavior for a set of templates defining a transformer, by defining two special named templates: `st:start` and `st:default`. The `st:start` template, if any, is selected at the beginning of the transformation process when no focus node is available. In that case, it is the first template executed by the template engine. The `st:default` template, if any, is executed when all templates fail to match the focus node.

The processing of a variable in the TEMPLATE clause by default consists in outputting its value in the Turtle format. A specific template, `st:profile`, can be used to overload this default transformation behaviour. For example, the following definition of `st:profile` specifies that processing any variable, denoted by `st:process(?x)`, consists in the application of the `st:apply-templates` function to it and that, in case no template applies, its default processing, denoted by `st:default(?x)`, should be the output of its string value, denoted by `str(?x)`, instead of the by default Turtle syntax, output by `st:turtle(?x)`.

```
TEMPLATE st:profile {
  st:define( st:process(?x) =
    st:apply-templates(?x) )
  st:define( st:default(?x) = str(?x) ) }
WHERE { }
```

5 Validation with Specific RDF Transformers

In our approach of RDF transformation based on SPARQL templates, the template processor is completely generic: it applies to any RDF data or any language or model provided with an RDF syntax. What is specific to each output language or format is the set of transformation rules defined for it. In other words, each transformer specific to an output format is a specific set of templates processed by the generic template processor implementing STTL.

In this section we present specific transformers available online[10] which validate both STTL and our implementation of a generic STTL processor. The applications of STTL are many and varied. A first family of applications deals with the presentation of RDF data into specific syntaxes, e.g., Turtle, (see Section 5.1), or presentation formats, e.g., HTML (see Section 5.2), or any other format answering specific

[10] http://ns.inria.fr/sparql-template

needs. As a result, STTL answers all the application scenarii addressed in the related work. A second family of applications deals with the transformation of statements of a given language represented in RDF syntax, e.g., OWL (see Section 5.3), or any other special purpose language with an RDF syntax. A third family of applications deals with the translation of RDF into other languages, e.g., RDF-to-CSV (see Section 5.2), or any translation X-to-Y of languages with RDF syntaxes.

5.1 RDF-to-RDF/Turtle Transformer

The following single STTL template enables to output RDF data in Turtle syntax.

```
TEMPLATE {
  ?x "\n"
  GROUP { ?p " " ?y ; separator = ";\n" }
  "." }
WHERE { ?x ?p ?y }
GROUP BY ?x
```

In a similar way, it is easy to write a transformer for each of RDF syntaxes.

5.2 RDF to HTML

We implemented a generic transformation to translate SPARQL query results into HTML. CONSTRUCT query returns an RDF graph whereas SELECT query result is translated in RDF using W3C DAWG result-set RDF vocabulary[11] which is a RDF version of SPARQL Query Results XML format.

The template below generates table cells for variable bindings:

```
prefix rs:
<http://www.w3.org/2001/sw/
DataAccess/tests/result-set#>
template  {
  "<td>"
  coalesce(
    st:call-template(st:display, ?val),
    " ")
  "</td>" ; separator = " "
}
where {
  ?x rs:solution ?in
  ?x rs:resultVariable ?var
  optional {
    ?in rs:binding [
      rs:variable ?var ; rs:value ?val ]
  }
}
order by ?var
```

Such RDF to HTML transformation enabled us to design a light weight Semantic Web Hypertext Navigator[12] on top of a local dataset as well as a remote dataset such as DBpedia. The transformer is embedded in a Web server, that is a SPARQL endpoint augmented with a transformation engine. The transformation engine is accessible at a specific URI on the server. Given a resource URI, the transformation retrieves a description of the resource in the dataset and generates a HTML page accordingly. In the HTML page, references to related resource URI are displayed as hypertext links to the Web server.

5.3 OWL 2 Pretty-Printer

We wrote a transformation generating OWL 2 expressions in functional syntax from OWL 2 expressions in RDF syntax as a set of 48 STTL templates.

We validated it on the OWL 2 Primer complete sample ontology[13] containing 350 RDF triples. To validate the result of the transformation, we loaded the output produced in OWL Functional syntax into Protégé and did a complete cycle of transformation (save to RDF/XML, load and transform again) and we checked that the results were equivalent. Let us note that the results are equivalent and not identical because some statements are not printed in the same order, due to the fact that Protégé does not save RDF/XML statements exactly in the same order and hence blank nodes are not allocated in the same order.

We tested this OWL/RDF transformer on several real world ontologies, among which a subset of the *Galen* ontology. The RDF graph representing it contains 33080 triples, the size of the result is 0.58 MB and the (average) transformation time is 1.75 seconds. We also have tested our pretty-printer on the *HAO* ontology. The RDF graph representing it contains 38842 triples, the size of the result is 1.63 MB, the (average) pretty-print time is 3.1 seconds.

In addition to the transformation of an RDF dataset representing OWL statements, this transformer can also be used when querying an OWL ontology stored in its RDF syntax, to present the results to the user in OWL 2 Functional syntax. This is done by calling in the SELECT clause of the query one of the extension functions launching the transformer. As an example, the following query retrieves specific classes of the ontology and displays them in Functional syntax:

[11] http://www.w3.org/2001/sw/DataAccess/tests/result-set

[12] http://corese.inria.fr

[13] http://www.w3.org/TR/owl2-primer

```
SELECT
  (st:apply-templates-with(st:owl, ?x)
  as ?t)
WHERE {
  ?x a owl:Class ;
     rdfs:subClassOf* f:Human }
```

5.4 Example of an Entire STTL Transformation

In this section we detail the execution of the OWL transformation[14] on the following OWL/RDF statement:

```
a:Parent owl:equivalentClass [
  a owl:Restriction ;
  owl:onProperty a:hasChild ;
  owl:someValuesFrom a:Person
]
```

The transformation engine first searches a template that matches the `owl:equivalentClass` statement. Here is the retrieved template:

```
TEMPLATE {
  if (bound(?t), "DatatypeDefinition",
    "EquivalentClasses")
  "("
    if (?iu,
      st:call-template(st:interunion, ?in),
      ?in)
    " " ?y
  ")"
}
WHERE {
  ?in owl:equivalentClass ?y
  BIND (EXISTS {
    ?in owl:intersectionOf|owl:unionOf ?z }
  as ?iu)
  OPTIONAL { ?y a ?t
    FILTER(?t = rdfs:Datatype) }
}
```

The TEMPLATE clause is compiled to:

```
SELECT
(concat(
  if (bound(?t), "DatatypeDefinition",
    "EquivalentClasses"),
  "(",
  if (?iu,
    st:call-template(st:interunion, ?in),
    st:process(?in)),
  " ", st:process(?y),
  ")")
as ?out)
```

[14] http://ns.inria.fr/sparql-template/owl

The WHERE clause of this template succeeds with the following bindings, where `_:b` is the blank node of type `owl:Restriction`:

```
  ?in = a:Parent ;
  ?y  = _:b ;
  ?iu = false .
```

The TEMPLATE clause then evaluates its arguments:

- The `?t` variable is not bound, hence the first expression evaluates to `"EquivalentClasses"`.
- The `?iu` variable value is false, hence the second expression evaluates `st:process(?in)`.

The following SELECT clause is eventually evaluated:
`SELECT (concat("EquivalentClasses", "(", st:process(?in), " ", st:process(?y), ")") as ?out).`
`st:process(?in)` with `?in` bound to IRI `a:Parent` then returns `a:Parent`. `st:process(?y)` with `?y` bound to blank node `_:b` of type `owl:Restriction`, subject of properties `owl:onProperty` and `owl:someValuesFrom`, then searches a template that matches such statements. Here is the retrieved template:

```
TEMPLATE {
  if (bound(?t1) || bound(?t2),
    "DataSomeValuesFrom",
    "ObjectSomeValuesFrom")
  "(" ?p " " ?z ")"
}
WHERE {
  ?in owl:someValuesFrom ?z ;
    owl:onProperty ?p
  OPTIONAL { ?z a ?t1
    FILTER(?t1 = rdfs:Datatype) }
  OPTIONAL { ?p a ?t2
    FILTER(?t2 = owl:DatatypeProperty) }
}
```

The TEMPLATE clause is compiled to:

```
SELECT
(concat(
  if (bound(?t1) || bound(?t2),
    "DataSomeValuesFrom",
    "ObjectSomeValuesFrom"),
  "(", st:process(?p), " ", st:process(?z), ")"
as ?out)
```

The WHERE clause of this template succeeds with the following bindings:

```
  ?in = _:b ;
  ?z  = a:Person ;
  ?p  = a:hasChild
```

The TEMPLATE clause of the above template then evaluates its arguments: variables `?t1` and `?t2` are not bound, hence the first expression evaluates to `"ObjectSomeValuesFrom"`.
The following SELECT clause is eventually evaluated:

```
SELECT (concat("ObjectSomeValuesFrom",
"(", st:process(?p), " ", st:process(?z),
")") as ?out).
```

As `?p` and `?z` both are bound to URIs, the evaluation of `st:process(?p)` and `st:process(?z)` eventually returns the Turtle format of these URI. The result of the above SELECT clause and therefore of the template is then the string:

```
"ObjectSomeValuesFrom(a:hasChild a:Person)"
```

As there is only one result, the final `group_concat(?out)` aggregate does not change it. This result is returned to the first template as the value of `st:process(?y)` in its SELECT clause . The result of this SELECT clause and therefore of the first template is then the following string:

```
"EquivalentClasses(a:Parent
ObjectSomeValuesFrom(a:hasChild a:Person))"
```

This is precisely the expression in OWL Functional syntax of the example of OWL statement chosen as input to illustrate the STTL transformation.

6 Conclusion and Future Work

In this paper we considered two related problems: (1) the transformation of RDF to present RDF data to users, e.g., into a HTML domain or application dependant format, and (2) the transformation of RDF when it is used as a meta-model to represent on the Web other languages and their abstract graph structure. We addressed the general problem of transforming RDF into other languages. We answered this question by specifying STTL, a generic and domain independent extension to SPARQL to support the declarative representation of any special-purpose RDF transformation as a set of transformation rules. Being based on SPARQL, STTL inherits its expressivity and its extension mechanisms. This specification and the algorithms we described have been implemented and tested in a generic transformation rule engine part of the Corese Semantic Web Factory platform (Corby et al., 2012; Corby and Faron-Zucker, 2010). This means all these results are part of this open-source platform. We demonstrated the feasibility and genericity of our approach by providing several transformations including: RDF-to-RDF syntaxes, RDF-to-HTML, RDF OWL 2-to-OWL 2 Functional syntax.

As future work, regarding the performances of our generic transformation rule engine, we intend to improve them by implementing heuristics to optimize the selection of templates. We should compare in the short term the performance of our generic transformation rule engine with that of existing tools for specific RDF transformations. For instance, we may compare the performance of our engine with that of the parser of the well known OWL API[15] for transforming large OWL 2 ontologies from RDF/XML syntax into Functional syntax.

Regarding the exploitation of our generic transformation rule engine to implement RDF transformers into specific languages, we intend to augment the number of STTL transformations available by writing rule sets for other formats and domains. In the cases where the TEMPLATE clauses of the transformation rules produce RDF triples (as text), we define RDF-to-RDF transformations with SPARQL Template. In particular, we envisage implementing a special case of RDF-to-RDF transformation to anonymize RDF datasets.

REFERENCES

Alkhateeb, F. and Laborie, S. (2008). Towards Extending and Using SPARQL for Modular Document Generation. In *Proc. of the 8th ACM Symposium on Document Engineering*, pages 164–172, Sao Paulo, Brésil. ACM Press.

Bischof, S., Decker, S., Krennwallner, T., Lopes, N., and Polleres, A. (2012). Mapping between RDF and XML with XSPARQL. *J. Data Semantics*, 1(3):147–185.

Bizer, C., Lee, R., and Pietriga, E. (2005). Fresnel - A Browser-Independent Presentation Vocabulary for RDF. In *Second International Workshop on Interaction Design and the Semantic Web @ ISWC'05*, Galway, Ireland.

Brophy, M. and Heflin, J. (2009). OWL-PL: A Presentation Language for Displaying Semantic Data on the Web. Technical report, Department of Computer Science and Engineering, Lehigh University.

Connolly, D. (2007). Gleaning Resource Descriptions from Dialects of Languages (GRDDL). Recommendation, W3C. http://www.w3.org/TR/grddl/.

Corby, O. and Faron-Zucker, C. (2010). The KGRAM Abstract Machine for Knowledge Graph Querying. In *IEEE/WIC/ACM International Conference*, Toronto, Canada.

Corby, O., Gaignard, A., Faron-Zucker, C., and Montagnat, J. (2012). KGRAM Versatile Data Graphs Querying and Inference Engine. In *Proc. IEEE/WIC/ACM International Conference on Web Intelligence*, Macau.

[15] http://owlapi.sourceforge.net/

Cyganiak, R., Wood, D., and Lanthaler, M. (2014). RDF 1.1 Concepts and Abstract Syntax. Recommendation, W3C. http://www.w3.org/TR/rdf11-concepts/.

Follenfant, C., Corby, O., Gandon, F., and Trastour, D. (2012). RDF Modelling and SPARQL Processing of SQL Abstract Syntax Trees. In *Programming the Semantic Web, ISWC Workshop*, Boston, USA.

Harris, S. and Seaborne, A. (2012). SPARQL 1.1 Query Language. Recommendation, W3C. http://www.w3.org/TR/sparql11-query/.

Horridge, M. and Bechhofer, S. (2011). The OWL API: A java API for OWL ontologies. *Semantic Web*, 2(1):11–21.

Kay, M. (2007). XSL Transformations (XSLT) Version 2.0. Recommendation, W3C. http://www.w3.org/TR/xslt20/.

Knublauch, H. (2011). SPIN - SPARQL Syntax. Member Submission, W3C. http://www.w3.org/Submission/2011/SUBM-spin-sparql-20110222/.

Peter F. Patel-Schneider and Motik, B. (2012). OWL 2 Web Ontology Language Mapping to RDF Graphs (Second Edition). Recommendation, W3C. http://www.w3.org/TR/owl-mapping-to-rdf/.

Quan, D. (2005). Xenon: An RDF Stylesheet Ontology. In *Proc. WWW*.

Robie, J., Chamberlin, D., Dyck, M., and Snelson, J. (2014). XQuery 3.0: An XML Query Language. Recommendation, W3C. http://www.w3.org/TR/xquery-30/.

Sandro Hawke, A. P. (2012). RIF In RDF. Working Group Note, W3C. http://www.w3.org/TR/rif-in-rdf/.

Stolz, A., Rodriguez-Castro, B., and Hepp, M. (2013). RDF Translator: A RESTful Multi-Format Data Converter for the Semantic Web. Technical report, E-Business and Web Science Research Group.