SPARQL Template: A Transformation Language for RDF

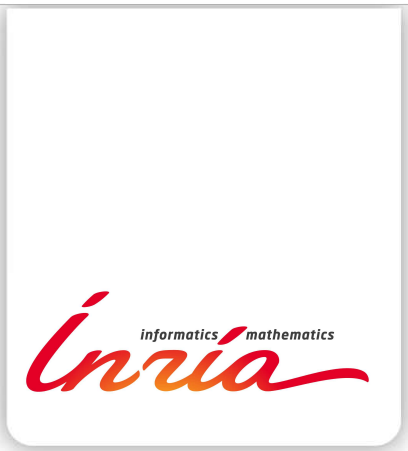
Olivier Corby, Catherine Faron Zucker, Fabien Gandon

SPARQL Template: A Transformation Language for RDF

Olivier Corby, Catherine Faron-Zucker, Fabien Gandon

RESEARCH REPORT
N° 8514
April 2014
Project-Teams Wimmics

ISSN 0249-6399 ISRN INRIA/RR--8514--FR+ENG



SPARQL Template: A Transformation Language for RDF

Olivier Corby*, Catherine Faron-Zucker†, Fabien Gandon ‡

Project-Teams Wimmics

Research Report n° 8514 — April 2014 — 22 pages

Abstract: RDF can be viewed as a meta-model to represent on the Web other languages and models, and in particular their abstract graph structure. The general research question addressed in this document is *How to transform RDF into other languages* and, in particular, how to generate the concrete syntax of expressions of a given language from their RDF representation. We show how SPARQL can be used as a generic RDF transformation rule language, independant from the output language. We define an RDF transformer as a set of transformation rules processed by a generic transformation rule engine. We present a lightweight syntactic extension to SPARQL in order to facilitate the writing of transformation rules and an implementation of a generic transformation rule engine. We show the feasability of our approach with several RDF transformers we have defined for various output languages.

Key-words: Semantic Web, SPARQL Template, RDF, Transformation, Pretty-Printing

* Inria, I3S
† Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271
‡ Inria, I3S

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

SPARQL Template: A Transformation Language for RDF

Résumé : RDF peut être considéré comme un méta modèle pour représenter sur le Web d'autres languages ou modèles et particulièrement leur structure abstraite sous forme de graphe. Ce document traite la question de recherche suivante : *Comment transformer un énoncé RDF dans un autre langage* et, en particulier, comment engendrer la syntaxe concrète d'énoncés d'un langage représenté en RDF (par exemple de OWL/RDF vers la syntaxe fonctionnelle de OWL).

Nous montrons comment SPARQL peut être utilisé comme un langage générique de règles de transformations, indépendant du format de sortie. Nous définissons une opération de transformation de RDF comme un ensemble de règles de transformation dédiées, interprétées par un moteur de transformation générique.

Nous présentons une extension syntaxique de SPARQL, qui facilite l'écriture des règles de transformation, et une implémentation de moteur de transformation générique. Enfin nous montrons la faisabilité de notre approche en présentant plusieurs transformation RDF réalisées pour différents langages.

Mots-clés : Web sémantique, SPARQL Template, RDF, Transformation, Pretty-Printing

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. Humans and machines can share this giant clip-board to push and pull data on a global scale. The RDF standard [7] provides us with a general purpose graph-oriented data model recommended by the W3C. It is provided with several syntaxes (XML, Turtle, JSON-LD, TriG, N-Triples, N-Quads, RDFa) 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, we believe that a special case of data holds a very special potential that has not been fully acknowledge yet: data encoding formal languages.

In mathematics, computer science, and other domains, many formal languages are created to capture information in symbolic representations constrained by rules. 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.

For instance in the domain of the Semantic Web alone, OWL 2 [12] 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) [14] 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 [11] to represent SPARQL rules in RDF, to facilitate storage and maintenance.

The Web is already used to exchange code and other representations of formal languages, but most of the time as documents. Just like the Semantic Web proposes to make data more accessible to machines by going beyond the initial documentary metaphor of the Web, we consider that the Web of data can be applied to formal language representations. These languages, their alphabets, their rules, their syntaxes and the expressions they produce can also be seen as just a special type of data. As such they can become the subject of data models and representations in other languages and, in particular in our case, of Web formalisms to exchange and even translate and transform them on a world-wide scale.

We propose to consider what happens when RDF is viewed and used as a meta-model to represent on the Web other languages and models, and in particular their abstract graph structure. We consider that RDF can then be viewed as a pivot abstract language to represent the Abstract Syntax Trees (AST) of expressions of other languages. To do so requires then the ability to export and import expressions between the targeted formal languages and their RDF representation. In this paper we address the latter problem of pretty-printing 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 into other languages?* We answer three sub-questions: (1) How

to write declarative transformation rules from RDF to other languages? (2) How to make the approach generic, i.e the rule language independant from the output language? and (3) Can SPARQL be used as a transformation rule language?

We show how SPARQL [9] can be used as a generic transformation rule language for RDF, independant from the output languages. We define a transformer as a set of transformation rules processed by a generic transformation rule engine. We present a lightweight syntactic extension to SPARQL in order to facilitate the writing of transformation rules and an implementation of a generic rule engine.

In section 2 we present existing transformation languages for Web languages. In section 3 we present a generic approach for writing RDF transformers, based on SPARQL as a transformation rule language. In section 4 we present the *template* syntactic extension to SPARQL facilitating the writing of RDF transformation rules. In section 5 we present our generic transformation rule engine. In section 6 we show the feasability of our approach with several RDF transformers we have defined for various output languages.

2 Related Work

Many specific languages have been designed to transform trees, among which PPML [15] was a pionner rule-based language for programming languages. It was designed in the context of the *Centaur* [3] programming environment.

XSLT [10] is another rule-based language for transforming XML trees. An XPath expression identifies the subtrees (their roots) for which the template applies, and the content of the template describes the text to be pretty-printed. Let us note that XSLT could be used to process and display RDF/XML data. For instance the following example is one of the XSLT rules (called *templates*) which could be used to pretty-print 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 serialization of an RDF statement and the transformation rules would depend on the concrete XML syntax of RDF instead of its semantics.

We have identified a few other specific transformation languages designed for the Semantic Web. Fresnel [2] 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. Lenses specify which properties of an RDF resource are displayed and how these properties are ordered; formats specify how resources and properties are rendered. The following example is 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.

```
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 .
```

Xenon [13] 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. Contrary to Fresnel formats, Xenon templates are tied to a specific display paradigm and an XHTML-like output format.

OWL-PL [4] 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. Like Xenon, OWL-PL is tied to its output format: XHTML.

[1] 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. Here again, the solution is specific to one output format, in this case XML.

In the following, we present a *generic* approach for writing RDF transformers for any output language, as opposed to the above presented works constrained by specific output languages. We propose a generic transformation rule language based on SPARQL and a generic transformation rule engine enabling to write

specific transformers for RDF: a special-purpose RDF transformer for a specific output language is declaratively defined as a specific set of transformation rules which can be processed by our generic rule engine.

3 SPARQL as a Transformation Rule Language

Generally speaking, a transformation rule consists in a *condition* part specifying statements to be transformed and a *presentation* part specifying the output for the statements matching the condition.

We propose to use the SPARQL language as a transformation rule language for RDF. We consider the SELECT-WHERE query form, with a WHERE clause as the condition part of a rule, enabling to select nodes in the RDF graph to be transformed, and a SELECT clause as the presentation part of the rule enabling to define the display of the selected nodes.

The WHERE clause of a transformation rule is any standard SPARQL graph pattern. Therefore, anything that can be represented in RDF can be transformed by such rules. For instance, let us consider the expressions in both Functional and RDF/Turtle syntaxes of a OWL 2 statement which defines the class of parents as equivalent to the class of individuals that are linked to a person by the `hasChild` property.

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

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

To pretty-print the above RDF statement into the corresponding Functional statement, the following WHERE clause matches the restriction expression and enables to select the values of properties `owl:onProperty` and `owl:allValuesFrom` and bind them to variables `?p` and `?c`[1].

```
where { ?in a owl:Restriction ;
            owl:onProperty ?p ;
            owl:someValuesFrom ?c .
}
```

[1]The actual pretty-print rule distinguishes between datatype and object properties; it is given in section 6.3.

The SELECT clause of a transformation rule produces an output result by using the values bound to the variables of the WHERE clause (or their display), combined with additional text. This is done by using the SPARQL `concat` function and an extension function we defined to recursively call transformation rules on the bound values: `st:apply-templates` is a recursive function which takes an RDF term as argument (we call it the focus node) and returns the result of the appropriate transformation rule applied to this node. For instance, here is the SELECT clause of the pretty-print rule for the OWL restriction in our running example.

```
prefix st: <http://ns.inria.fr/sparql-template/>
select
   (st:apply-templates(?p) as ?pp)
   (st:apply-templates(?c) as ?pc)
   (concat("ObjectSomeValuesFrom(",
       ?pp, " ", ?pc, ")") as ?out)
```

Variables `?pp` and `?pc` are the transformation results for variables `?p` and `?c`. The pretty-printing of the *Restriction* statement as a whole is the concatenation of appropriate text and the values bound to variables `?pp` and `?pc`. It is returned by the SELECT clause as the value of variable `?out`.

By convention, the focus node of a transformation rule (in the WHERE clause) is denoted by `?in` and the transformation result (in the SELECT clause) is denoted by `?out`. Thus, the result of the `st:apply-templates` function with the value bound to `?in` as argument is the value bound to `?out` and the transformer is the implementation of `st:apply-templates`. This is detailed in section 5.

By analogy with XSLT templates, we call *templates* our SPARQL SELECT-WHERE transformation rules. In the next section, we present a syntax for writing templates.

4 Extension of SPARQL with the template-where Query Form

SPARQL Template is a lightweight syntactic extension to SPARQL we defined to facilitate the writing of transformation rules. It consists in a TEMPLATE-WHERE query form which has some similarity with the CONSTRUCT-WHERE query form but generates text (in any format) instead of RDF triples.

4.1 Template Definition

SPARQL Template defines and uses the namespace shown below:

```
prefix st: <http://ns.inria.fr/sparql-template/>
```

Here is the syntax of the TEMPLATE extension, based on SPARQL 1.1 Query Language grammar[2]. `Prologue` defines base and prefixes, `PrimaryExpression`

[2] http://www.w3.org/TR/sparql11-query/

is a constant, a variable, a function call or a bracketed expression.

```
Template ::= Prologue
   'template' (iri VarList ?) ? '{'
   (PrimaryExpression | Group) *
   ( Separator ) ?
   '}'
   WhereClause
   SolutionModifier
   ValuesClause

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

Group ::= 'group' ('distinct') ? '{'
   PrimaryExpression *
   ( Separator ) ?
   '}'

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

4.2 Template Compiling

Compiling a template keeps the WHERE clause, the solution modifiers and the VALUES clause of the template unchanged. Only the TEMPLATE clause is modified which is syntactic sugar for `st:apply-templates()` and `(concat () as ?out)`. Here is the compiling scheme of a TEMPLATE clause into a standard SELECT clause. Basically, a recursive `tr` function replaces variables `V` by `st:apply-templates(V)` and it concatenates terms in a `concat()` (1-4). Other expressions are left as is (5-6).

```
(1) tr(template(List l)) -> select (concat(tr(l)) as ?out)
(2) tr(List l)           -> List(tr(e)) for all e in l
(3) tr(Group(List l))    -> group_concat(tr(l))
(4) tr(Variable v)       -> coalesce(st:apply-templates(v), "")
(5) tr(Literal d)        -> d
(6) tr(Exp f)            -> f
```

For instance, the following template:

```
template {
  "ObjectAllValuesFrom(" ?p " " ?c ")"
}
where {
  ?in a owl:Restriction ;
    owl:onProperty ?p ;
    owl:allValuesFrom ?c .
}
```

is compiled into the following standard SELECT-WHERE query:

```
select
  (coalesce(st:apply-templates(?p), "") as ?_v1)
  (coalesce(st:apply-templates(?c), "") as ?_v2)
  (concat(
    "ObjectAllValuesFrom(", ?_v1, " ", ?_v2, ")")
    as ?out)
where {
  ?in a owl:Restriction ;
     owl:onProperty ?p ;
     owl:allValuesFrom ?c .
}
```

Variables are compiled into `coalesce(st:apply-templates(var), "")` in order to facilitate the writing of templates with `optional` or `union` patterns where variables may be unbound. Hence, for unbound variables, the empty string is returned in the pretty-print stream.

4.3 Transformation Functions in the template Clause

SPARQL functions can be used in the template clause, e.g. `xsd:string(term)` returns the string value of its argument. In addition, we defined a set of specific transformation functions that can be used in the TEMPLATE clause of a transformation rule:

- `st:apply-templates(term)` calls the transformer on a focus node and executes one template.
- `st:apply-templates-with(uri, term)` calls the specified transformer on a focus node and executes one template. The `uri` argument is the URI of the document that defines the templates.
- `st:apply-all-templates(term)` calls the transformer on a focus node and executes all templates; it returns the concatenation of the results.
- `st:call-template(name, term)` calls a template by its name on a focus node, it can have several node arguments.
- `st:call-template-with(uri, name, term)` calls a template by its name, on a focus node with a specified transformer.
- `st:turtle(term)` returns the Turtle form of an RDF term.
- `st:uri(term)` returns the Turtle form of its argument if it is a URI; else calls `st:apply-templates`.

For instance, the following template:

```
template {
  "list: " group { ?exp }
}
where {
  ?in rdf:rest*/rdf:first ?exp
}
```

is compiled into the following standard SPARQL SELECT-WHERE query:

```
select
  (coalesce(st:apply-templates(?exp), "") as ?res)
  (group_concat(?res) as ?p)
  (concat("list: ", ?p) as ?out)
where {
  ?in rdf:rest*/rdf:first ?exp
}
```

5 A SPARQL Template Processor as a Generic RDF Transformer

As stated in section 3 the RDF transformer is the `st:apply-templates` extension function. It is implemented together with the TEMPLATE extension to SPARQL in the Corese Semantic Web Factory[3] [6, 5] .

Given an RDF graph with a focus node to be transformed and a list of templates, the `st:apply-templates` function 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. Here is the core pseudocode of `st:apply-templates`.

```
(01) Node st:apply-templates(Node node){
(02)   for (Query q : getTemplates()){
(03)     Mappings map = eval(q, IN, node);
(04)     Node res = map.getResult(OUT);
(05)     if (res != null) return res;
(06)   }
(07)   return node;
(08) }
```

Templates are selected (02) and tried (03) one by one until one of them returns a result (04-05). In other words, a template is searched whose WHERE clause matches the RDF graph with the binding of the focus node. If no template is found, the focus node itself is returned (07).

Recursive calls to `st:apply-templates` implements the AST recursive traversal with successive focus nodes: `eval` (03) runs templates that recursively call the `st:apply-templates` function.

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

In addition to this pseudocode, the transformer checks loops in case the RDF graph is not a tree but a cyclic graph. It keeps track of the templates applied to nodes in order to avoid applying the same template on the same node twice. If no fresh template exists for a focus node, the transformer returns its string value.

5.1 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)
```

It can be generalized to named templates with arguments:

```
template st:rec(?x) {
}
where {
   bind(st:getFocusNode('x') as ?x)
}
```

5.2 Template Selection

Named templates are processed explicitly by a call to the `st:call-template()` function. For instance, the following template calls the `st:interunion` template.

```
template {
 "SubClassOf("
    if (bound(?z),
      st:call-template(st:interunion, ?in), st:uri(?in))
    " "  st:uri(?y) ")"
}
where {
?in rdfs:subClassOf ?y
   optional {
     {?in owl:intersectionOf ?z} union
     {?in owl:unionOf ?z}}
}
```

Given a focus node, the template processor considers the first template that matches this node. Hence, the result of the transformation of the focus node is the result of this template. It is possible to specify an order between templates using an explicit *priority*. For this purpose we introduce a PRAGMA extension to the syntax:

```
template {}
where    {}
pragma   { st:template st:priority 1 }
```

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-all-templates()` function in the TEMPLATE clause. The result of the transformation is the concatenation of the results of the successful templates, possibly with a user-defined separator.

```
template {
  st:apply-all-templates(?in
    ; separator = "\n")
}
where {...}
```

A transformer can be used to transform a whole RDF graph. For this purpose, the `st:apply-templates-with(uri)` function can be called without focus node. The transformer must then determine a focus node as there is no distinguished root node in a graph. In this case, if there is a `st:start` named template, it is executed first. Otherwise, the first template that succeeds is the starting point of the transformer.

5.3 Recursive Templates

By combining recursive calls of named templates and `if` patterns, our transformer is able to perform computations. For instance, the following two templates enable to generate the development of $n!$ (the first one calling the second one).

```
template {
  ?n "! = " st:call-template(st:rec, ?n)
}
where {
  ?in a m:Factorial ; m:args(?n)
}

template st:rec(?x) {
  if (?x = 1, 1, concat(?x, " . ",
     st:call-template(st:rec, ?x - 1)))
}
where { }
```

5.4 Managing Several Transformation Results

In the general case, a template may return several results, like any standard SELECT-WHERE query. In that case, the transformer automatically concatenates these results. Let Q be the SELECT-WHERE query resulting from the compilation of a template T. Let M be the multiset of solution mappings that is the final result of the evaluation of Q. The result of the evaluation of T is computed by applying an additional `group_concat` aggregate on variable `?out` on multiset M as shown below, where `scalarvals` is a separator that may be user-defined.

```
Aggregation((?out), group_concat, scalarvals, M)
```

As a result, the values of the `?out` variable of the final results are concatenated, after SPARQL processing of eventual aggregates and solution modifiers.

An explicit call to the `group_concat` function in the TEMPLATE clause can also be done to specify how to aggregate transformation results. For instance, the following template displays authors and their documents (separated by the *space* default separator in the explicit call to `group_concat`), grouped by authors (separated by the specified line break separator in the implicit final call to `group_concat`):

```
template {
  "author: " ?a "\n"
  "books: " group_concat(?b)
   ; separator = "\n"
}
where {
  ?in ex:member ?a
  ?a  ex:author ?b
}
group by ?a
order by ?a
```

6 Validation with Several Specific RDF Transformers

In our approach of RDF transformation based on SPARQL templates with a SPARQL external function `st:apply-templates`, the transformer, i.e. the template processor is completely generic: it applies to any RDF data or any language provided with an RDF syntax. What is specific to each output language or format is the set of transformation rules defined for it.

We have written several SPARQL template bases available online[4] which validate both SPARQL Template and our generic transformer approach.

[4] ftp://ftp-sop.inria.fr/wimmics/soft/pprint

6.1 RDF-to-RDF Transformer

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

```
template {
  st:uri(?x) "\n"
  group { st:uri(?p) " " st:turtle(?y)
    ; separator = ";\n" } "."
  ; separator = " \n\n"
}
where { ?x ?p ?y }
group by ?x
```

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

6.2 RDF-to-HTML and RDF-to-CSV Transformer

We implemented a generic transformer from RDF to HTML which generates a HTML table with properties in lines and resources in columns. The content of a cell is the value(s) of a given property for a given resource. Here is the template that generates the HTML code for the cells containing RDF triple values:

```
template st:line {
  "<td>"
  group {
    if (?o = st:null, "",
    if (isURI(?o),
      st:call-template(st:href, ?o),
      replace(
         replace(st:turtle(?o), "&", "&amp;"),
         "<", "&lt;")))
  }
  "</td>"
}
where {
  {select distinct ?p where {?s ?p ?v}
   order by ?p}
  {?in ?p ?o} union {bind(st:null as ?o)}
}
group by ?p
order by ?p
```

With a simple adaptation, it is easy to write as well a transformer to output a CSV table representing an RDF graph.

6.3 OWL 2 Pretty-Printer

We wrote a pretty-printer generating OWL 2 expressions in functional syntax from OWL 2 expressions in RDF syntax as a set of 48 SPARQL templates. Among them, here are the two templates enabling to pretty-print the OWL/RDF statements presented in section 3 in OWL 2 Functional syntax.

```
template {
  if (bound(?t), "DatatypeDefinition",
    "EquivalentClasses") "("
  if (bound(?z),
    st:call-template(st:interunion, ?in),
    st:uri(?in))
  " "  st:uri(?y) ")"
}
where {
  ?in owl:equivalentClass ?y
  optional {{?in owl:intersectionOf ?z}
    union {?in owl:unionOf ?z}}
  optional { ?y a ?t .
    filter(?t = rdfs:Datatype)}
}

template {
  if (bound(?t1) || bound(?t2),
    "DataSomeValuesFrom",
    "ObjectSomeValuesFrom")
  "(" st:uri(?p) " " st:uri(?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)}
}
```

We validated it on the OWL 2 Primer complete sample ontology[5] containing 350 RDF triples. To validate the result of the pretty-printer, we loaded the output produced in OWL Functional syntax into Protégé and did a complete cycle of pretty print (save to RDF/XML, load and pretty print 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

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

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 0.56 MB, the number of template calls is 291476 and the (average) pretty-print time is 1.75 seconds.

6.4 SPIN Pretty-Printer

SPIN provides an RDF syntax for SPARQL queries. Here is an example of a SPARQL query in SPIN syntax:

```
[] a  sp:Select ;
   sp:resultVariables (_:b1 _:b2) ;
   sp:where (
     [ sp:subject _:b1 ;
       sp:predicate foaf:name ;
       sp:object _:b2 ]
     [a sp:ne ;
       sp:arg1 _:b2 ;
       sp:arg2 "James"]) .
_:b1  sp:varName "x"^^xsd:string .
_:b2  sp:varName "name"^^xsd:string .
```

The result of the transformation in SPARQL concrete syntax is shown below:

```
select ?x ?name
where {
?x foaf:name ?name .
  filter(?name != "James")
}
```

We defined a pretty-printer for SPIN (SPARQL 1.1 Query & Update) as a set of 62 SPARQL templates. It generates SPARQL 1.1 queries in concrete syntax from RDF SPIN data. Here are some key templates of our pretty-printer.

```
template {
  "construct "
  if (bound(?temp),
    concat("{", ?temp, "}", "\n"), "")
  "where {" ?where "}"
}
where {
  ?in a sp:Construct ; sp:where ?where
  optional { ?in sp:templates ?temp }
}
```

```
template {
  st:uri(?x) " " st:uri(?p) " " st:uri(?y) "."
}
where {
  ?in sp:subject ?x ;
    sp:predicate ?p ;
    sp:object ?y
}
```

We tested our SPIN pretty-printer with success on 444 queries from the W3C SPARQL 1.1 Query & Update test cases. We compiled in SPIN the SPARQL queries in concrete syntax in the W3C test cases. Then we converted this RDF data back into SPARQL queries in concrete syntax with our SPIN pretty-printer.

6.5 SQL Pretty-Printer

We proposed an RDF syntax for SQL queries similar to the RDF syntax of SPARQL queries in SPIN in [8]. Here is an extract of an RDF AST for a SQL select statement:

```
@prefix sql: <http://ns.inria.fr/ast/sql#>
[] a sql:Query ;
  sql:args (
   [a sql:Select ;
    sql:args (
     [a sql:Column ;
      sql:label "Customers.CustomerName"]
     [a sql:Column ;
      sql:label "Orders.OrderID"] ) ]
   ... )
```

We have implemented a SQL pretty-printer from SQL AST in RDF to SQL concrete syntax. Here is a template outputing the select statement of a SQL query from its RDF AST:

```
prefix sql:<http://ns.inria.fr/ast/sql#>
template  {
  "select "
  st:call-template(sql:comma, ?select)
}
where {
  ?in a sql:Select ; sql:args ?select
}
```

6.6 RDF-to-Latex Transformer

We made an experiment with mathematical expressions represented in RDF. We defined an RDF AST for mathematical expressions and we wrote a set

of SPARQL templates to transform it into Latex. Here is an example of a mathematical expression that we encoded in RDF:

```
[a m:Integral ;
  m:from 0 ; m:to m:pi ;
  m:args (
  [a m:Integral ;
  m:from 0 ; m:to m:pi ;
  m:args(
  [a m:Plus ; m:args(
    [a m:Mult ; m:args(3
      [a m:Mult ; m:args([m:name "x"]
        [m:name "y"])] )]
    [a m:Plus ; m:args(
      [a m:Mult ; m:args(2 [m:name "x"] )]
      1
    )]
  )]
  ) ;
  m:var  [m:name "x"]] ) ;
  m:var  [m:name "y"] ]
```

Here is the Latex format generated by our transformer:

```
\int_{y=0}^{\pi}\,\int_{x=0}^{\pi}\,
  (3 \, x \, y + 2 \, x + 1)\,dx\,dy
```

And here is the final output produced by a standard Latex compiler:

$$\int_{y=0}^{\pi} \int_{x=0}^{\pi} (3\,x\,y + 2\,x + 1)\,dx\,dy$$

The following templates are involved in the transformation of the above example. Here is a template to pretty-print an integral:

```
template {
  "\\int_{" ?v "=" ?fr "}^{" ?to "}\\, "
    if (bound(?p), "(", "")
    ?e
    if (bound(?p), ")", "")
    "\\,"
  "d" ?v
}
where {
  ?in a m:Integral ; m:args (?e) ;
   m:var ?v ;
   m:from ?fr ; m:to ?to
```

```
  optional {
    ?e a ?p .
    filter(?p = m:Plus) }
}
```

And here is a template to pretty-print a term:

```
template {
  ?f " " ?t " " ?r }
where {
  ?in a ?t ; m:args(?f ?r)
}
values ?t {
  m:Plus m:Minus m:Mult
  m:Eq m:Lt m:Le m:Ne m:Gt m:Ge
}
```

6.7 OWL2RL-to-SPARQL Transformer

We made an experiment on the transformation of OWL 2 RL statements into SPARQL CONSTRUCT-WHERE queries expressing rules. For instance, here is a template that translates a OWL 2 property chain axiom into a SPARQL rule with a property path expression generated on-the-fly.

```
template {
  "construct {?x " st:turtle(?in) " ?y } \n"
  "where {"
     "?x "
     group { st:turtle(?e) ; separator = "/" }
     " ?y }"
}
where {
  ?in owl:propertyChain ?l .
  ?l rdf:rest*/rdf:first ?e
}
```

Given the following property chain axiom:

```
ex:hasUncle owl:propertyChain
  (ex:hasFather ex:hasBrother)
```

the above template generates the following SPARQL rule:

```
construct { ?x ex:hasUncle ?y }
where { ?x ex:hasFather/ex:hasBrother ?y }
```

6.8 Discussion

The applications of our work are many and varied. A first family of applications deals with the presentation of RDF data into specific syntaxes (Turtle, RDF/XML, RDF/JSON, TriG, N-Triples, N-Quads, RDFa) or presentation formats (HTML table or any other format answering specific needs). As a result, we are able to answer all the application scenarii adressed in the related work.

A second family of applications deals with the translation of RDF into other languages: RDF-to-CSV, RDF-to-Conceptual Graphs, RDF-to-Kif, etc. but also the translation X-to-Y of other languages with RDF syntaxes (e.g. OWL2RL-to-SPARQL).

A third family of applications deals with the pretty-print of statements of a given language stored in RDF syntax: OWL, SPARQL, SQL, or any other special purpose language with an RDF syntax. For instance, when querying an OWL ontology represented in its RDF syntax, SPARQL queries can call the OWL 2 pretty-printer in their SELECT clause to present their results in the Functional syntax. For instance, the following query retrieves specific classes of the ontology and displays them in Functional syntax in the result:

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

In a variant of this application scenario, the RDF expression of the statements of some language is embedded into heterogenous RDF data. In that case, the pretty-printer for the RDF AST of this specific language just handles a subset of the whole RDF data.

Note that the transformation engine can also be used in a filter clause in order to perform string text matching:

```
select *
where {
  ?x a owl:Class
  filter(contains(st:apply-templates-with(st:owl, ?x),
     "intersection"))
}
```

7 Conclusion and On-going Work

In this article we proposed to consider that RDF can be used as a meta-model to represent on the Web other languages and their abstract graph structure. Among the research questions that derive from this proposal we addressed the problem of transforming RDF into other languages in particular to translate, transform or export these expressions to many other formats. We specified

a generic and domain independent extension to support SPARQL Templates formalizing the transformation rules. Being based on SPARQL, the template language 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 [6, 5]. This means all these results are part of the open-source platform Corese. We demonstrated the feasibility and genericity of our approach by providing several transformations including: RDF-to-RDF syntaxes, OWL 2, SPIN, HTML, SQL, Latex. We now intend to augment the number of transformations available by writing rules for other formats and domains. We also consider special applications of this generic transformation mechanism including for instance anonymizing dumps or rewriting heuristics for optimization.

Acknowledgment

We thank Abdoul Macina and Corentin Follenfant for their work on the SQL-to-RDF compiler and the RDF-to-SQL pretty-printer.

References

[1] Faisal Alkhateeb and Sébastien Laborie. 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, September 2008. ACM Press.

[2] Christian Bizer, Ryan Lee, and Emmanuel Pietriga. Fresnel - A Browser-Independent Presentation Vocabulary for RDF. In *Second International Workshop on Interaction Design and the Semantic Web @ ISWC'05*, Galway, Ireland, November 2005.

[3] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the system. In *Proc. SIGSOFT, 3rd Annual Symposium on Software Development Environments*, Boston, USA, 1988.

[4] Matt Brophy and Jeff Heflin. OWL-PL: A Presentation Language for Displaying Semantic Data on the Web. Technical report, Department of Computer Science and Engineering, Lehigh University, 2009.

[5] Olivier Corby and Catherine Faron-Zucker. The KGRAM Abstract Machine for Knowledge Graph Querying. In *IEEE/WIC/ACM International Conference*, Toronto, Canada, September 2010.

[6] Olivier Corby, Alban Gaignard, Catherine Faron-Zucker, and Johan Montagnat. KGRAM Versatile Data Graphs Querying and Inference Engine. In *Proc. IEEE/WIC/ACM International Conference on Web Intelligence*, Macau, December 2012.

[7] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. Recommendation, W3C, 2014. http://www.w3.org/TR/rdf11-concepts/.

[8] Corentin Follenfant, Olivier Corby, Fabien Gandon, and David Trastour. RDF Modelling and SPARQL Processing of SQL Abstract Syntax Trees. In *Programming the Semantic Web, ISWC Workshop*, Boston, USA, November 2012.

[9] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. Recommendation, W3C, 2013. http://www.w3.org/TR/sparql11-query/.

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

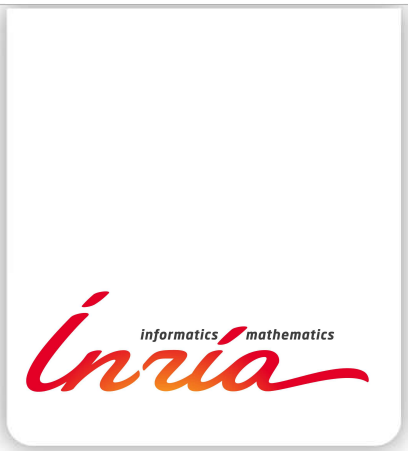
[11] Holger Knublauch. SPIN - SPARQL Syntax. Member Submission, W3C, 2011. http://www.w3.org/Submission/2011/SUBM-spin-sparql-20110222/.

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

[13] Dennis Quan. Xenon: An RDF Stylesheet Ontology. In *Proc. WWW*, 2005.

[14] Axel Polleres Sandro Hawke. RIF In RDF. Working Group Note, W3C, 2012. http://www.w3.org/TR/rif-in-rdf/.

[15] Laurent Théry. A Table-Driven Compiler for Pretty Printing Specifications. Technical Report RT 0288, Inria, 2003. http://hal.inria.fr/docs/00/06/98/91/PDF/RT-0288.pdf.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399