XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage*

Waseem Akhtar[1], Jacek Kopecký[2], Thomas Krennwallner[1], and Axel Polleres[1]

[1] Digital Enterprise Research Institute, National University of Ireland, Galway
{firstname.lastname}@deri.org
[2] STI Innsbruck, University of Innsbruck, Austria
jacek.kopecky@uibk.ac.at

Abstract. With currently available tools and languages, translating between an existing XML format and RDF is a tedious and error-prone task. The importance of this problem is acknowledged by the W3C GRDDL working group who faces the issue of extracting RDF data out of existing HTML or XML files, as well as by the Web service community around SAWSDL, who need to perform lowering and lifting between RDF data from a semantic client and XML messages for a Web service. However, at the moment, both these groups rely solely on XSLT transformations between RDF/XML and the respective other XML format at hand. In this paper, we propose a more natural approach for such transformations based on merging XQuery and SPARQL into the novel language XSPARQL. We demonstrate that XSPARQL provides concise and intuitive solutions for mapping between XML and RDF in either direction, addressing both the use cases of GRDDL and SAWSDL. We also provide and describe an initial implementation of an XSPARQL engine, available for user evaluation.

1 Introduction

There is a gap within the Web of data: on one side, XML provides a popular format for data exchange with a rapidly increasing amount of semi-structured data available. On the other side, the Semantic Web builds on data represented in RDF, which is optimized for data interlinking and merging; the amount of RDF data published on the Web is also increasing, but not yet at the same pace. It would clearly be useful to enable reuse of XML data in the RDF world and vice versa. However, with currently available tools and languages, translating between XML and RDF is not a simple task.

The importance of this issue is currently being acknowledged within the W3C in several efforts. The Gleaning Resource Descriptions from Dialects of Languages [9] (GRDDL) working group faces the issue of extracting RDF data out of existing (X)HTML Web pages. In the Semantic Web Services community, RDF-based client software needs to communicate with XML-based Web services, thus it needs to perform transformations between its RDF data and the XML messages that are exchanged with the Web

* This material is based upon works supported by the European FP6 projects inContext (IST-034718) and TripCom (IST-4-027324-STP), and by Science Foundation Ireland under Grant No. SFI/02/CE1/I131.

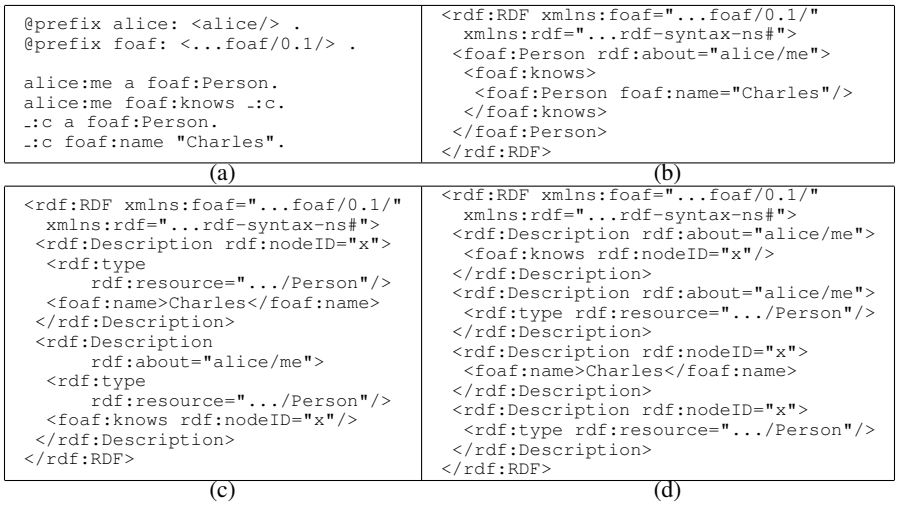


Fig. 1. Different representations of the same RDF graph

services. The Semantic Annotations for WSDL (SAWSDL) working group calls these transformations *lifting* and *lowering* (see [12,14]). However, both these groups propose solutions which rely solely on XSL transformations (XSLT) [10] between RDF/XML [2] and the respective other XML format at hand. Using XSLT for handling RDF data is greatly complicated by the flexibility of the RDF/XML format. XSLT (and XPath) were optimized to handle XML data with a simple and known hierarchical structure, whereas RDF is conceptually different, abstracting away from fixed, tree-like structures. In fact, RDF/XML provides a lot of flexibility in how RDF graphs can be serialized. Thus, processors that handle RDF/XML as XML data (not as a set of triples) need to take different possible representations into account when looking for pieces of data. This is best illustrated by a concrete example: Fig. 1 shows four versions of the same FOAF (cf. `http://www.foaf-project.org`) data.[3] The first version uses Turtle [3], a simple and readable textual format for RDF, inaccessible to pure XML processing tools though; the other three versions are all RDF/XML, ranging from concise (b) to verbose (d).

The three RDF/XML variants look very different to XML tools, yet exactly the same to RDF tools. For any variant we could create simple XPath expressions that extract for instance the names of the persons known to Alice, but a single expression that would correctly work in all the possible variants would become more involved. Here is a list of particular features of the RDF data model and RDF/XML syntax that complicate XPath+XSLT processing:

- Elements denoting properties can directly contain value(s) as nested XML, or reference other descriptions via the `rdf:resource` or `rdf:nodeID` attributes.
- References to resources can be relative or absolute URIs.
- Container membership may be expressed as `rdf:li` or `rdf:_1`, `rdf:_2`, etc.
- Statements about the same subject do not need to be grouped in a single element.

[3] In listings and figures we often abbreviate well-known namespace URIs (`http://www.w3.org/1999/02/22-rdf-syntax-ns#`, `http://xmlns.com/foaf/0.1/`, etc.) with "...".

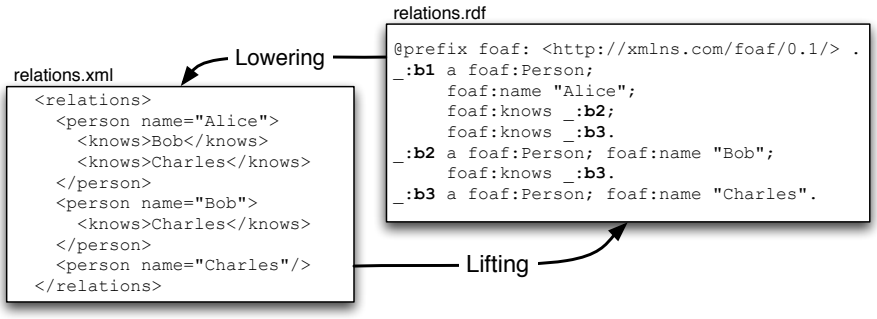


Fig. 2. From XML to RDF and back: "lifting" and "lowering"

- String-valued property values such as `foaf:name` in our example (and also values of `rdf:type`) may be represented by XML element content or as attribute values.
- The type of a resource can be represented directly as an XML element name, with an explicit `rdf:type` XML element, or even with an `rdf:type` attribute.

This is not even a complete list of the issues that complicate the formulation of adequate XPath expressions that cater for every possible alternative in how one and the same RDF data might be structured in its concrete RDF/XML representation.

Apart from that, simple reasoning (e.g., RDFS materialization) improves data queries when accessing RDF data. For instance, in FOAF, every Person (and Group and Organization etc.) is also an Agent, therefore we should be able to select all the instances of `foaf:Agent`. If we wanted to write such a query in XPath+XSLT, we literally would need to implement an RDFS inference engine within XSLT. Given the availability of RDF tools and engines, this seems to be a dispensable exercise.

Recently, two new languages have entered the stage for processing XML and RDF data: XQuery [5] is a W3C Recommendation since early last year and SPARQL [20] has finally received W3C's Recommendation stamp in January 2008. While both languages operate in their own worlds – SPARQL in the RDF- and XQuery in the XML-world – we show in this paper that the merge of both in the novel language XSPARQL has the potential to finally bring XML and RDF closer together. XSPARQL provides concise and intuitive solutions for mapping between XML and RDF in either direction, addressing both the use cases of GRDDL and SAWSDL. As a side effect, XSPARQL may also be used for RDF to RDF transformations beyond the capabilities of "pure" SPARQL. We also describe an implementation of XSPARQL, available for user evaluation.

In the following, we elaborate a bit more in depth on the use cases of lifting and lowering in the contexts of both GRDDL and SAWSDL in Section 2 and discuss how they can be addressed by XSLT alone. Next, in Section 3 we describe the two starting points for an improved lifting and lowering language – XQuery and SPARQL – before we announce their happy marriage to XSPARQL in Section 4. Particularly, we extend XQuery's **FLWOR** expressions with a way of iterating over SPARQL results. We sketch the semantics of XSPARQL and describe a rewriting algorithm that translates XSPARQL to XQuery. By this we can show that XSPARQL is a conservative extension of both XQuery and SPARQL. A formal treatment of XSPARQL showing this correspondence

```
<xsl:stylesheet
  xmlns:xsl="...XSL/Transform"
  xmlns:foaf="...foaf/0.1/"
  xmlns:rdf="...rdf-syntax-ns#"
  version="2.0">

<xsl:template match="/relations">
  <rdf:RDF>
    <xsl:apply-templates />
  </rdf:RDF>
</xsl:template>

<xsl:template match="person">
  <foaf:Person>
    <foaf:name>
      <xsl:value-of
         select="./@name"/>
    </foaf:name>
    <xsl:apply-templates/>
  </foaf:Person>
</xsl:template>

<xsl:template match="knows">
  <foaf:knows><foaf:Person>
    <foaf:name>
      <xsl:apply-templates/>
      </foaf:name>
  </foaf:Person></foaf:knows>
</xsl:template>

</xsl:stylesheet>
```

(a) mygrddl.xsl

```
<rdf:RDF xmlns:rdf="...rdf-syntax-ns#"
         xmlns:foaf="...foaf/0.1/">
<foaf:Person>
  <foaf:name>Alice</foaf:name>
  <foaf:knows><foaf:Person>
      <foaf:name>Bob</foaf:name>
  </foaf:Person></foaf:knows>
  <foaf:knows><foaf:Person>
      <foaf:name>Charles</foaf:name>
  </foaf:Person></foaf:knows>
</foaf:Person>
<foaf:Person>
  <foaf:name>Bob</foaf:name>
  <foaf:knows><foaf:Person>
      <foaf:name>Charles</foaf:name>
  </foaf:Person></foaf:knows>
</foaf:Person>
<foaf:Person>
  <foaf:name>Charles</foaf:name>
</foaf:Person>
</rdf:RDF>
```

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
_:b1 a foaf:Person; foaf:name "Alice";
     foaf:knows _:b2; foaf:knows _:b3.
_:b2 a foaf:Person; foaf:name "Bob".
_:b3 a foaf:Person; foaf:name "Charles".
_:b4 a foaf:Person; foaf:name "Bob";
     foaf:knows _:b5 .
_:b5 a foaf:Person; foaf:name "Charles" .
_:b6 a foaf:Person; foaf:name "Charles".
```

(b) Result of the GRDDL transform in RDF/XML (up) and Turtle (down)

Fig. 3. Lifting attempt by XSLT

is given in an extended version of this paper [1]. We wrap up the paper with an outlook to related and future works and conclusions to be drawn in Section 5 and 6.

2 Motivation – Lifting and Lowering

As a running example throughout this paper we use a mapping between FOAF data and a customized XML format as shown in Fig. 2. The task here in either direction is to extract for all persons the names of people they know. In order to keep things simple, we use element and attribute names corresponding to the respective classes and properties in the FOAF vocabulary (i.e., `Person`, `knows`, and `name`). We assume that names in our XML file uniquely identify a person which actually complicates the transformation from XML to RDF, since we need to create a unique, distinct blank node per name. The example data is a slight variant of the data from Fig. 1, where Alice knows both Bob and Charles, Bob knows Charles, and all parties are identified by blank nodes.

Because semantic data in RDF is on a higher level of abstraction than semi-structured XML data, the translation from XML to RDF is often called "lifting" while the opposite direction is called "lowering," as also shown in Fig. 2.

Lifting in GRDDL. The W3C Gleaning Resource Descriptions from Dialects of Languages (GRDDL) working group has the goal to complement the concrete RDF/XML syntax with a mechanism to relate to other XML dialects (especially XHTML or "microformats") [9]. GRDDL focuses on the lifting task, i.e., extracting RDF from XML. To this end, the working group recently published a finished Recommendation which

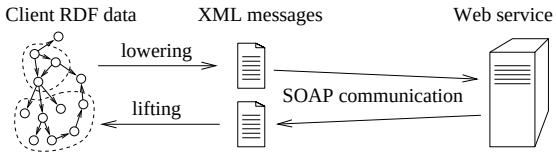


Fig. 4. RDF data lifting and lowering for WS communication

specifies how XML files or XML Schema namespace documents can reference transformations that are then processed by a GRDDL-aware application to extract RDF from the respective source file. Typically – due to its wide support – XSLT [10] is the language of choice to describe such transformations. However, writing XSLT can be cumbersome, since it is a general-purpose language for producing XML without special support for creating RDF. For our running example, the XSLT in Fig. 3(a) could be used to generate RDF/XML from the relations.xml file in Fig. 2 in an attempt to solve the lifting step. Using GRDDL, we can link this XSLT file mygrddl.xsl from relations.xml by changing the root element of the latter to:

```
<relations xmlns:grddl="http://www.w3.org/2003/g/data-view#"
           grddl:transformation="mygrddl.xsl"> ...
```

The RDF/XML result of the GRDDL transformation is shown in the upper part of Fig. 3(b). However, if we take a look at the Turtle version of this result in the lower part of Fig. 3(b) we see that this transformation creates too many blank nodes, since this simple XSLT does not merge equal names into the same blank nodes.

XSLT is a Turing-complete language, and theoretically any conceivable transformation can be programmed in XSLT; so, we could come up with a more involved stylesheet that creates unique blank node identifiers per name to solve the lifting task as intended. However, instead of attempting to repair the stylesheet from Fig. 3(a) let us rather ask ourselves whether XSLT is the right tool for such transformations. The claim we make is that specially tailored languages for RDF-XML transformations like XSPARQL which we present in this paper might be a more suitable alternative to alleviate the drawbacks of XSLT for the task that GRDDL addresses.

Lifting/Lowering in SAWSDL. While GRDDL is mainly concerned with lifting, in SAWSDL (Semantic Annotations for WSDL and XML Schema) there is a strong need for translations in the other direction as well, i.e., from RDF to arbitrary XML.

SAWSDL is the first standardized specification for semantic description of Web services. Semantic Web Services (SWS) research aims to automate tasks involved in the use of Web services, such as service discovery, composition and invocation. However, SAWSDL is only a first step, offering hooks for attaching semantics to WSDL components such as operations, inputs and outputs, etc. Eventually, SWS shall enable client software agents or services to automatically communicate with other services by means of semantic mediation on the RDF level. The communication requires both lowering and lifting transformations, as illustrated in Fig. 4. Lowering is used to create the request XML messages from the RDF data available to the client, and lifting extracts RDF from the incoming response messages.

```
<xsl:stylesheet version="1.0" xmlns:rdf="...rdf-syntax-ns#"
  xmlns:foaf="...foaf/0.1/" xmlns:xsl="...XSL/Transform">
<xsl:template match="/rdf:RDF">
  <relations><xsl:apply-templates select=".//foaf:Person"/></relations>
</xsl:template>
<xsl:template match="foaf:Person"><person name="./@foaf:name">
  <xsl:apply-templates select="./foaf:knows"/>
</person></xsl:template>
<xsl:template match="foaf:knows[@rdf:nodeID]"><knows>
   <xsl:value-of select="//foaf:Person[@rdf:nodeID=./@rdf:nodeID]/@foaf:name"/>
</knows></xsl:template>
<xsl:template match="foaf:knows[foaf:Person]">
  <knows><xsl:value-of select="./foaf:Person/@foaf:name"/></knows>
</xsl:template>
</xsl:stylesheet>
```

Fig. 5. Lowering attempt by XSLT (mylowering.xsl)

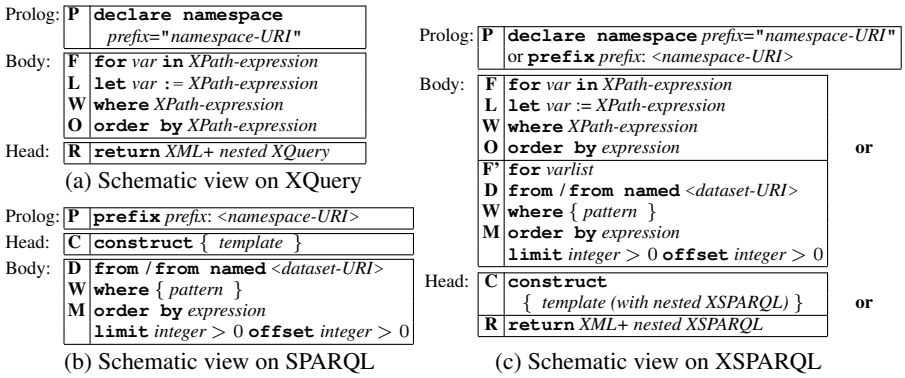
As opposed to GRDDL, which provides hooks to link XSLT transformations on the level of whole XML or namespace documents, SAWSDL provides a more fine-grained mechanism for "semantic adornments" of XML Schemas. In WSDL, schemata are used to describe the input and output messages of Web service operations, and SAWSDL can annotate messages or parts of them with pointers to relevant semantic concepts plus links to lifting and lowering transformations. These links are created using the `sawsdl:liftingSchemaMapping` and `sawsdl:loweringSchemaMapping` attributes which reference the transformations within XSL elements (`xsl:element`, `xsl:attribute`, etc.) describing the respective message parts.

SAWSDL's schema annotations for lifting and lowering are not only useful for communication with web services from an RDF-aware client, but for service mediation in general. This means that the output of a service S_1 uses a different message format than service S_2 expects as input, but it could still be used if service S_1 and S_2 provide lifting and lowering schema mappings, respectively, which map from/to the same ontology, or, respectively, ontologies that can be aligned via ontology mediation techniques (see [11]).

Lifting is analogous to the GRDDL situation – the client or an intermediate mediation service receives XML and needs to extract RDF from it –, but let us focus on RDF data lowering now. To stay within the boundaries of our running example, we assume a social network site with a Web service for querying and updating the list of a user's friends. The service accepts an XML format à la relations.xml (Fig. 2) as the message format for updating a user's (client) list of friends.

Assuming the client stores his FOAF data (relations.rdf in Fig. 2) in RDF/XML in the style of Fig. 1(b), the simple XSLT stylesheet mylowering.xsl in Fig. 5 would perform the lowering task. The service could advertise this transformation in its SAWSDL by linking mylowering.xsl in the `sawsdl:loweringSchemaMapping` attribute of the XML Schema definition of the `relations` element that conveys the message payload. However, this XSLT will break if the input RDF is in any other variant shown in Fig. 1. We could create a specific stylesheet for each of the presented variants, but creating one that handles all the possible RDF/XML forms would be much more complicated.

In recognition of this problem, SAWSDL contains a non-normative example which performs a lowering transformation as a sequence of a SPARQL query followed by an XSLT transformation on SPARQL's query results XML format [6]. Unlike XSLT or XPath, SPARQL treats all the RDF input data from Fig. 1 as equal. This approach makes a step in the right direction, combining SPARQL with XML technologies. The detour



(a) Schematic view on XQuery

(b) Schematic view on SPARQL

(c) Schematic view on XSPARQL

Fig. 6. An overview of XQuery, SPARQL, and XSPARQL

through SPARQL's XML query results format however seems to be an unnecessary burden. The XSPARQL language proposed in this paper solves this problem: it uses SPARQL pattern matching for selecting data as necessary, while allowing the construction of arbitrary XML (by using XQuery) for forming the resulting XML structures.

As more RDF data is becoming available on the Web which we want to integrate with existing XML-aware applications, SAWSDL will obviously not remain the only use case for lowering.

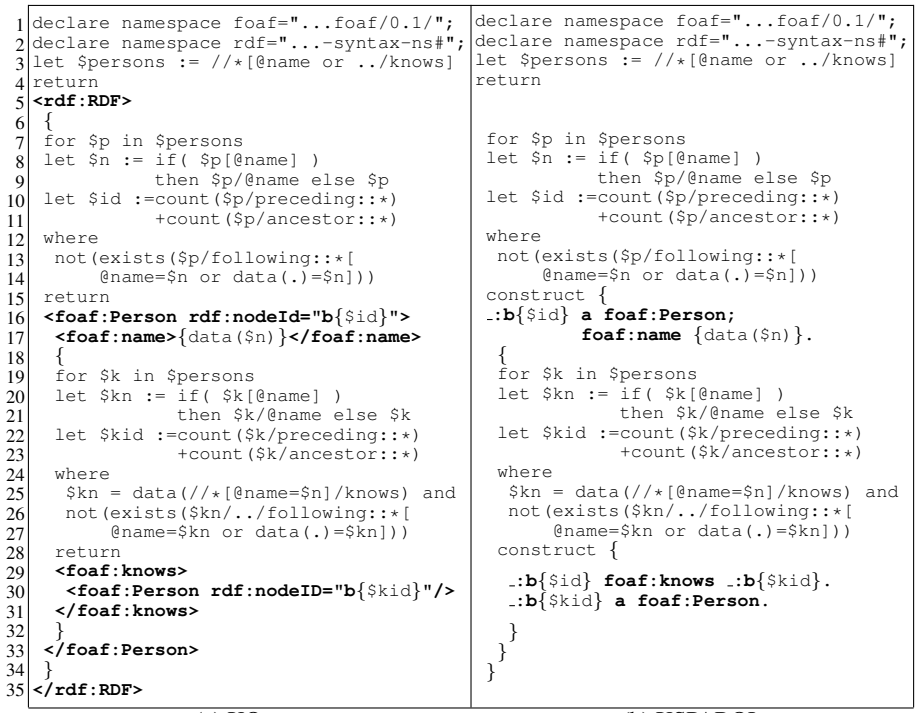
3 Starting Points: XQuery and SPARQL

In order to end up with a better suited language for specifying translations between XML and RDF addressing both the lifting and lowering use cases outlined above, we can build up on two main starting points: XQuery and SPARQL. Whereas the former allows a more convenient and often more concise syntax than XSLT for XML query processing and XML transformation in general, the latter is the standard for RDF querying and construction. Queries in each of the two languages can roughly be divided in two parts: (i) the retrieval part (*body*) and (ii) the result construction part (*head*). Our goal is to combine these components for both languages in a unified language, XSPARQL, where XQuery's and SPARQL's heads and bodies may be used interchangeably. Before we go into the details of this merge, let us elaborate a bit on the two constituent languages.

XQuery. As shown in Fig. 6(a) an XQuery starts with a (possibly empty) prolog (**P**) for namespace, library, function, and variable declarations, followed by so called **FLWOR** – or "flower" – expressions, denoting body (**FLWO**) and head (**R**) of the query. We only show namespace declarations in Fig. 6 for brevity.

As for the body, **`for`** clauses (**F**) can be used to declare variables looping over the XML nodeset returned by an XPath expression. Alternatively, to bind the entire result of an XPath query to a variable, **`let`** assignments can be used. The **`where`** part (**W**) defines an XPath condition over the current variable bindings. Processing order of results of a **`for`** can be specified via a condition in the **`order by`** clause (**O**).

In the head (**R**) arbitrary well-formed XML is allowed following the **`return`** keyword, where variables scoped in an enclosing **`for`** or **`let`** as well as nested XQuery **FLWOR** expressions are allowed.



```
1  declare namespace foaf="...foaf/0.1/";
2  declare namespace rdf="...-syntax-ns#";
3  let $persons := //*[@name or ../knows]
4  return
5  <rdf:RDF>
6   {
7   for $p in $persons
8   let $n := if( $p[@name] )
9             then $p/@name else $p
10  let $id :=count($p/preceding::*)
11            +count($p/ancestor::*)
12  where
13   not(exists($p/following::*[
14       @name=$n or data(.)=$n]))
15  return
16  <foaf:Person rdf:nodeId="b{$id}">
17   <foaf:name>{data($n)}</foaf:name>
18   {
19   for $k in $persons
20   let $kn := if( $k[@name] )
21              then $k/@name else $k
22   let $kid :=count($k/preceding::*)
23              +count($k/ancestor::*)
24   where
25    $kn = data(//*[@name=$n]/knows) and
26    not(exists($kn/../following::*[
27        @name=$kn or data(.)=$kn]))
28   return
29   <foaf:knows>
30    <foaf:Person rdf:nodeID="b{$kid}"/>
31   </foaf:knows>
32   }
33  </foaf:Person>
34  }
35 </rdf:RDF>
```

(a) XQuery

```
declare namespace foaf="...foaf/0.1/";
declare namespace rdf="...-syntax-ns#";
let $persons := //*[@name or ../knows]
return


 for $p in $persons
 let $n := if( $p[@name] )
           then $p/@name else $p
 let $id :=count($p/preceding::*)
           +count($p/ancestor::*)
 where
  not(exists($p/following::*[
      @name=$n or data(.)=$n]))
 construct {
 _:b{$id} a foaf:Person;
          foaf:name {data($n)}.
  {
  for $k in $persons
  let $kn := if( $k[@name] )
             then $k/@name else $k
  let $kid :=count($k/preceding::*)
             +count($k/ancestor::*)
  where
   $kn = data(//*[@name=$n]/knows) and
   not(exists($kn/../following::*[
       @name=$kn or data(.)=$kn]))
  construct {

   _:b{$id} foaf:knows _:b{$kid}.
   _:b{$kid} a foaf:Person.

  }
 }
}
```

(b) XSPARQL

Fig. 7. Lifting using XQuery and XSPARQL

Any XPath expression in **FLWOR** expressions can again possibly involve variables defined in an enclosing **`for`** or **`let`**, or even nested XQuery **FLWOR** expressions. Together with a large catalogue of built-in functions [15], XQuery thus offers a flexible instrument for arbitrary transformations. For more details, we refer the reader to [5,7].

The lifting task of Fig. 2 can be solved with XQuery as shown in Fig. 7(a). The resulting query is quite involved, but completely addresses the lifting task, including unique blank node generation for each person: We first select all nodes containing person names from the original file for which a blank node needs to be created in variable `$p` (line 3). Looping over these nodes, we extract the actual names from either the value of the `name` attribute or from the `knows` element in variable `$n`. Finally, we compute the position in the original XML tree as blank node identifier in variable `$id`. The **`where`** clause (lines 12–14) filters out only the last name for duplicate occurrences of the same name. The nested **`for`** (lines 19–31) to create nested `foaf:knows` elements again loops over persons, with the only differences that only those nodes are filtered out (line 25), which are known by the person with the name from the outer **`for`** loop.

While this is a valid solution for lifting, we still observe the following drawbacks: (1) We still have to build RDF/XML "manually" and cannot make use of the more readable and concise Turtle syntax; and (2) if we had to apply XQuery for the lowering task, we still would need to cater for all kinds of different RDF/XML representations. As we will see, both these drawbacks are alleviated by adding some SPARQL to XQuery.

```
prefix vc: <...vcard-rdf/3.0#>
prefix foaf: <...foaf/0.1/>
construct {$X foaf:name $FN.}
from <vc.rdf>
where { $X vc:FN $FN .}
```

(a)

```
prefix vc: <...vcard-rdf/3.0#>
prefix foaf: <...foaf/0.1/>
construct {_:b foaf:name
            {fn:concat("""",$N," ",$F,""""")}.}
from <vc.rdf>
where { $P vc:Given $N. $P vc:Family $F.}
```

(b)

Fig. 8. RDF-to-RDF mapping in SPARQL (a) and an enhanced mapping in XSPARQL (b)

```
<relations>{ for $Person $Name from <relations.rdf> where {$Person foaf:name $Name}
order by $Name return <person name="{$Name}">{
 for $FName from <relations.rdf> where
 {$Person foaf:knows $Friend. $Person foaf:name $Name. $Friend foaf:name $Fname}
 return <knows>{$FName}</knows> }</person> }</relations>
```

Fig. 9. Lowering using XSPARQL

SPARQL. Fig. 6(b) shows a schematic overview of the building blocks that SPARQL queries consist of. Again, we do not go into details of SPARQL here (see [20,17,18] for formal details), since we do not aim at modifying the language, but concentrate on the overall semantics of the parts we want to reuse. Like in XQuery, namespace prefixes can be specified in the Prolog (**P**). In analogy to **FLWOR** in XQuery, let us define so-called **DWMC** expressions for SPARQL.

The body (**DWM**) offers the following features. A *dataset* (**D**), i.e., the set of source RDF graphs, is specified in **`from`** or **`from named`** clauses. The **`where`** part (**W**) – unlike XQuery – allows to match parts of the dataset by specifying a graph *pattern* possibly involving variables, which we denote $\mathrm{vars}(pattern)$. This pattern is given in a Turtle-based syntax, in the simplest case by a set of triple patterns, i.e., triples with variables. More involved patterns allow unions of graph patterns, optional matching of parts of a graph, matching of named graphs, etc. Matching patterns on the conceptual level of RDF graphs rather than on a concrete XML syntax alleviates the pain of having to deal with different RDF/XML representations; SPARQL is agnostic to the actual XML representation of the underlying source graphs. Also the RDF merge of several source graphs specified in consecutive **`from`** clauses, which would involve renaming of blank nodes at the pure XML level, comes for free in SPARQL. Finally, variable bindings matching the **`where`** pattern in the source graphs can again be ordered, but also other solution modifiers (**M**) such as **`limit`** and **`offset`** are allowed to restrict the number of solutions considered in the result.

In the head, SPARQL's **`construct`** clause (**C**) offers convenient and XML-independent means to create an output RDF graph. Since we focus here on RDF construction, we omit the **`ask`** and **`select`** SPARQL query forms in Fig. 6(b) for brevity. A **`construct`** *template* consists of a list of triple patterns in Turtle syntax possibly involving variables, denoted by $\mathrm{vars}(template)$, that carry over bindings from the **`where`** part. SPARQL can be used as transformation language between different RDF formats, just like XSLT and XQuery for transforming between XML formats. A simple example for mapping full names from vCard/RDF (`http://www.w3.org/TR/vcard-rdf`) to `foaf:name` is given by the SPARQL query in Fig. 8(a).

Let us remark that SPARQL does not offer the generation of new values in the head which on the contrary comes for free in XQuery by offering the full range of

XPath/XQuery built-in functions. For instance, the simple query in Fig. 8(b) which attempts to merge family names and given names into a single `foaf:name` is beyond SPARQL's capabilities. As we will see, XSPARQL will not only make reuse of SPARQL for transformations from and to RDF, but also aims at enhancing SPARQL itself for RDF-to-RDF transformations enabling queries like the one in Fig. 8(b).

4 XSPARQL

Conceptually, XSPARQL is a simple merge of SPARQL components into XQuery. In order to benefit from the more intuitive facilities of SPARQL in terms of RDF graph matching for retrieval of RDF data and the use of Turtle-like syntax for result construction, we syntactically add these facilities to XQuery. Fig. 6(c) shows the result of this "marriage." First of all, every native XQuery query is also an XSPARQL query. However we also allow the following modifications, extending XQuery's **FLWOR** expressions to what we call (slightly abusing nomenclature) **FLWOR'** expressions: (i) In the body we allow SPARQL-style **F'DWM** blocks alternatively to XQuery's **FLWO** blocks. The new **F' `for`** clause is very similar to XQuery's native **`for`** clause, but instead of assigning a single variable to the results of an XPath expression it allows the assignment of a whitespace separated list of variables (*varlist*) to the bindings for these variables obtained by evaluating the graph pattern of a SPARQL query of the form: **`select`** *varlist* **DWM**. (ii) In the head we allow to create RDF/Turtle directly using **`construct`** statements (**C**) alternatively to XQuery's native **`return`** (**R**).

These modifications allows us to reformulate the lifting query of Fig. 7(a) into its slightly more concise XSPARQL version of Fig. 7(b). The real power of XSPARQL in our example becomes apparent on the lowering part, where all of the other languages struggle. Fig. 9 shows the lowering query for our running example.

As a shortcut notation, we allow also to write "**`for`** *" in place of "**`for`** [*list of all variables appearing in the* **`where`** *clause*]"; this is also the default value for the **F'** clause whenever a SPARQL-style **`where`** clause is found and a **`for`** clause is missing. By this treatment, XSPARQL is also a syntactic superset of native SPARQL **`construct`** queries, since we additionally allow the following: (1) XQuery and SPARQL namespace declarations (**P**) may be used interchangeably; and (2) SPARQL-style **`construct`** result forms (**R**) may appear before the retrieval part; note that we allow this syntactic sugar only for queries consisting of a single **FLWOR'** expression, with a single **`construct`** appearing right after the query prolog, as otherwise, syntactic ambiguities may arise. This feature is mainly added in order to encompass SPARQL style queries, but in principle, we expect the (**R**) part to appear in the end of a **FLWOR'** expression. This way, the queries of Fig. 8 are also syntactically valid for XSPARQL.

Semantics and Implementation. As we have seen above, XSPARQL syntactically subsumes both XQuery and SPARQL. Concerning semantics, XSPARQL equally builds on top of its constituent languages. In an earlier version of this paper [1], we have extended the formal semantics of XQuery [7] by additional rules which reduce each XSPARQL query to XQuery expressions; the resulting **FLWOR**s operate on the answers of SPARQL queries in the SPARQL XML result format [6]. Since we add only new

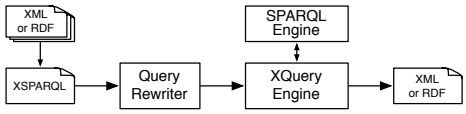


Fig. 10. XSPARQL architecture

reduction rules for SPARQL-like heads and bodies, it is easy to see that each native XQuery is treated in a semantically equivalent way in XSPARQL.

In order to convince the reader that the same holds for native SPARQL queries, we will illustrate our reduction in the following. We restrict ourselves here to a more abstract presentation of our rewriting algorithm, as we implemented it in a prototype.[4]

The main idea behind our implementation is translating XSPARQL queries to corresponding XQueries which possibly use interleaved calls to a SPARQL endpoint. The architecture of our prototype shown in Fig. 10 consists of three main components: (1) a query rewriter, which turns an XSPARQL query into an XQuery; (2) a SPARQL endpoint, for querying RDF from within the rewritten XQuery; and (3) an XQuery engine for computing the result document.

The rewriter (Alg. 1) takes as input a full XSPARQL *QueryBody* [7] q (i.e., a sequence of **FLWOR'** expressions), a set of bound variables b and a set of position variables p, which we explain below. For a **FL** (or **F'**, resp.) clause s, we denote by $\mathrm{vars}(s)$ the list of all newly declared variables (or the *varlist*, resp.) of s. For the sake of brevity, we only sketch the core rewriting function *rewrite*() here; additional machinery handling the prolog including function, variable, module, and namespace declarations is needed in the full implementation. The rewriting is initiated by invoking *rewrite*$(q, \emptyset, \emptyset)$ with empty bound and position variables, whose result is an XQuery. Fig. 11 shows the output of our translation for the **`construct`** query in Fig. 8(b) which illustrates both the lowering and lifting parts. Let us explain the algorithm using this sample output.

After generating the prolog (lines 1–9 of the output), the rewriting of the QueryBody is performed recursively following the syntax of XSPARQL. During the traversal of the nested **FLWOR'** expressions, SPARQL-like heads or bodies will be replaced by XQuery expressions, which handle our two tasks. The lowering part is processed first:

Lowering The lowering part of XSPARQL, i.e., SPARQL-like **F'DWM** blocks, is "encoded" in XQuery with interleaved calls to an external SPARQL endpoint. To this end, we translate **F'DWM** blocks into equivalent XQuery **FLWO** expressions which retrieve SPARQL result XML documents [6] from a SPARQL engine; i.e., we "push" each **F'DWM** body to the SPARQL side, by translating it to a native **`select`** query string. The auxiliary function *sparql*() in line 6 of our rewriter provides this functionality, transforming the **`where`** $\{pattern\}$ part of **F'DWM** clauses to XQuery expressions which have all bound variables in $\mathrm{vars}(pattern)$ replaced by the values of the variables; "free" XSPARQL variables serve as binding variables for the SPARQL query result. The outcome of the *sparql*() function is a list of expressions, which is concatenated and URI-encoded using XQuery's XPath functions, and wrapped into a URI with `http` scheme pointing to the SPARQL query service (lines 10–12 of the output), cf. [6]. Then we create a new XQuery **`for`** loop over variable `$aux_result` to iterate over the query

[4] `http://www.polleres.net/xsparql/`

```
1  declare vc = "http://www.w3.org/2001/vcard-rdf/3.0#";
2  declare foaf = "http://xmlns.com/foaf/0.1/";
3  declare sparql_result = "http://www.w3.org/2005/sparql-results#";
4  declare function local:rdf_term($NT as xs:string,$V as xs:string) as xs:string {
5   let $rdf_term := if($NT="literal") then fn:concat("""",$V,"""")
6    else if($NT="bnode") then fn:concat("_:",$V) else if($NT="uri")
7    then fn:concat("<",$V,">") else "" return $rdf_term };
8  declare variable $NS_1 := "prefix vc: <...vcard-rdf/3.0#> ";
9  declare variable $NS_2 := "prefix foaf: <...foaf/0.1/> ";
10 fn:concat("@",$NS_1,".","@",$NS_2,"."), let $aux_query := fn:concat(
11  "http://localhost:2020/sparql?query=", fn:encode-for-uri(fn:concat($NS_1, $NS_2,
12  "select $P $N $F from <vc.rdf> where {$P vc:Given $N. $P vc:Family $F.}")))
13 for $aux_result at $aux_result_pos in doc($aux_query)//sparql_result:result
14  let $P_Node := $aux_result/sparql_result:binding[@name="P"]
15  let $P := data($P_Node/*) let $P_NodeType := name($P_Node/*)
16  let $P_RDFTerm := local:rdf_term($P_NodeType,$P)
17  let $N_Node := $aux_result/sparql_result:binding[@name="N"]
18  let $N := data($N_Node/*) let $N_NodeType := name($N_Node/*)
19  let $N_RDFTerm := local:rdf_term($N_NodeType,$N)
20  let $F_Node := $aux_result/sparql_result:binding[@name="F"]
21  let $F := data($F_Node/*) let $F_NodeType := name($F_Node/*)
22  let $F_RDFTerm := local:rdf_term($F_NodeType,$F)
23 return ( fn:concat("_:b",$aux_result_pos," foaf:name "),
24         ( fn:concat("""",$N_RDFTerm," ",$F_RDFTerm,"""") ), "." )
```

Fig. 11. XQuery encoding of Example 8(b)

answers extracted from the SPARQL XML result returned by the SPARQL query processor (line 13). For each variable $\$x_i \in \mathrm{vars}(s)$ (i.e., in the (**F'**) **`for`** clause of the original **F'DWM** body), new auxiliary variables are defined in separate **`let`**-expressions extracting its node, content, type (i.e., literal, uri, or blank), and RDF-Term ($\$x_i$`_Node`, $\$x_i$, $\$x_i$`_NodeType`, and $\$x_i$`_RDFTerm`, resp.) by appropriate XPath expressions (lines 14–22 of Fig. 11); the *auxvars*() helper in line 6 of Alg. 1 is responsible for this.

Lifting For the lifting part, i.e., SPARQL-like **`construct`**s in the **R** part, the transformation process is straightforward. Before we rewrite the QueryBody q, we process the prolog (**P**) of the XSPARQL query and output every namespace declaration as Turtle string literals "`@prefix ns: <URI>`." (line 10 of the output). Then, Alg. 1 is called on q and recursively decorates every **`for`** $\$Var$ expression by fresh position variables (line 13 of our example output); ultimately, **`construct`** templates are rewritten to an assembled string of the pattern's constituents, filling in variable bindings and evaluated subexpressions (lines 23–24 of the output).

Blank nodes in **`construct`**s need special care, since, according to SPARQL's semantics, these must create new blank node identifiers for each solution binding. This is solved by "adorning" each blank node identifier in the **`construct`** part with the above-mentioned position variables from any enclosing **`for`**-loops, thus creating a new, unique blank node identifier in each loop (line 23 in the output). The auxiliary function *rewrite-template*() in line 8 of the algorithm provides this functionality by simply adding the list of all position variable p as expressions to each blank node string; if there are nested expressions in the supplied **`construct`** $\{template\}$, it will return a sequence of nested **FLWOR**s with each having *rewrite*() applied on these expressions with the in-scope bound and position variables.

Expressions involving **`construct`**s create Turtle output. Generating RDF/XML output from this Turtle is optionally done as a simple postprocessing step supported by using standard RDF processing tools.

Algorithm 1: *rewrite*(q, b, p) : Rewrite XSPARQL q to an XQuery

Input: XSPARQL query q, set of bounded variables b, set of position variables p
Result: XQuery

if q *is of form* s_1 `,` ... `,` s_k **then**
  return *rewrite*(s_1, b, p) `,` ... `,` *rewrite*(s_k, b, p)
else if q *is of form* `for` $\$x_1$ `in` $XPathExpr_1$`,` ... `,` $\$x_k$ `in` $XPathExpr_k$ s_1 **then**
  return `for` $\$x_1$ `at` $\$x_1$*_pos* `in` $XPathExpr_1$`,` ... `,` $\$x_k$ `at` $\$x_k$*_pos* `in` $XPathExpr_k$ *rewrite*$(s_1, b, p \cup \{\$x_1\text{-}pos, \ldots, \$x_k\text{-}pos\})$
else if q *is of form* `for` $\$x_1 \cdots \$x_n$ `from` $\mathbf{D}$ `where` $\{$ *pattern* $\}$ $\mathbf{M}$ s_1 **then**
  return `let` $\$aux\_query$ `:=` *sparql*$(\mathbf{D}, \{\$x_1, \ldots, \$x_n\}$, *pattern*, $\mathbf{M}$, b `)` `for` $\$aux\_result$ `in` `doc(` $\$aux\_query$ `)//sparql:result` *auxvars*$(\{\$x_1, \ldots, \$x_n\})$ *rewrite*$(s_1, b \cup \text{vars}(q), p)$
else if q *is of form* `construct` $\{template\}$ **then**
  return `return (` *rewrite-template*$(template, b, p)$ `)`
else
  split q into its subexpressions $s_1, \ldots, s_n$
  for $j := 1, \ldots, n$ **do** $b_j = b \cup \bigcup_{1 \le i \le j-1} \text{vars}(s_i)$
  if $n > 1$ **then return** $q\,[s_1/$*rewrite*$(s_1, b_1, p), \ldots, s_n/$*rewrite*$(s_n, b_n, p)]$
  else return q
end

5 Related Works

Albeit both XML and RDF are nearly a decade old, there has been no serious effort on developing a language for convenient transformations between the two data models. There are, however, a number of apparently abandoned projects that aim at making it easier to transform RDF data using XSLT. *RDF Twig* [21] suggests XSLT extension functions that provide various useful views on the "sub-trees" of an RDF graph. The main idea of RDF Twig is that while RDF/XML is hard to navigate using XPath, a subtree of an RDF graph can be serialized into a more useful form of RDF/XML. *TreeHugger*[5] makes it possible to navigate the graph structure of RDF both in XSLT and XQuery using XPath-like expressions, abstracting from the actual RDF/XML structure. *rdf2r3x*[6] uses an RDF processor and XSLT to transform RDF data into a predictable form of RDF/XML also catering for RSS. Carroll and Stickler take the approach of simplifying RDF/XML one step further, putting RDF into a simple *TriX* [4] format, using XSLT as an extensibility mechanism to provide human-friendly macros for this syntax.

These approaches rely on non-standard extensions or tools, providing implementations in some particular programming language, tied to specific versions of XPath or XSLT processors. In contrast, *RDFXSLT*[7] provides an XSLT preprocessing stylesheet and a set of helper functions, similar to RDF Twig and TreeHugger, yet implemented in pure XSLT 2.0, readily available for many platforms.

[5] `http://rdfweb.org/people/damian/treehugger/index.html`
[6] `http://wasab.dk/morten/blog/archives/2004/05/30/transforming-rdfxml-with-xslt`
[7] `http://www.wsmo.org/TR/d24/d24.2/v0.1/20070412/rdfxslt.html`

All these proposals focus on XPath or XSLT, by adding RDF-friendly extensions, or preprocessing the RDF data to ease the access with stock XPath expressions. It seems that XQuery and SPARQL were disregarded previously because XQuery was not standardized until 2007 and SPARQL – which we suggest to select relevant parts of RDF data instead of XPath – has only very recently received W3C's recommendation stamp.

As for the use of SPARQL, Droop et al. [8] suggest, orthogonal to our approach, to compile XPath queries into SPARQL. Similarly, encoding SPARQL completely into XSLT or XQuery [13] seems to be an interesting idea that would enable to compile down XSPARQL to pure XQuery without the use of a separate SPARQL engine. However, scalability results in [13] so far do not yet suggest that such an approach would scale better than the interleaved approach we took in our current implementation.

Finally, related to our discussion in Section 2, the SPARQL Annotations for WSDL (SPDL) project (`http://www.w3.org/2005/11/SPDL/`) suggests a direct integration of SPARQL queries into XML Schema, but is still work in progress. We expect SPDL to be subsumed by SAWSDL, with XSPARQL as the language of choice for lifting and lowering schema mappings.

6 Conclusion and Future Plans

We have elaborated on use cases for lifting and lowering, i.e., mapping back and forth between XML and RDF, in the contexts of GRDDL and SAWSDL. As we have seen, XSLT turned out to provide only partially satisfactory solutions for this task. XQuery and SPARQL, each in its own world, provide solutions for the problems we encountered, and we presented XSPARQL as a natural combination of the two as a proper solution for the lifting and lowering tasks. Moreover, we have seen that XSPARQL offers more than a handy tool for transformations between XML and RDF. Indeed, by accessing the full library of XPath/XQuery functions, XSPARQL opens up extensions such as value-generating built-ins or even aggregates in the construct part, which have been pointed out missing in SPARQL earlier [19].

As we have seen, XSPARQL is a conservative extension of both of its constituent languages, SPARQL and XQuery. The semantics of XSPARQL was defined as an extension of XQuery's formal semantics adding a few normalization mapping rules. We provide an implementation of this transformation which is based on reducing XSPARQL queries to XQuery with interleaved calls to a SPARQL engine via the SPARQL protocol. There are good reasons to abstract away from RDF/XML and rely on native SPARQL engines in our implementation. Although one could try to compile SPARQL entirely into an XQuery that caters for all different RDF/XML representations, that would not solve the use which we expect most common in the nearer future: many online RDF sources will most likely not be accessible as RDF/XML files, but rather via RDF stores that provide a standard SPARQL interface.

Our resulting XSPARQL preprocessor can be used with any available XQuery and SPARQL implementation, and is available for user evaluation along with all examples and an extended version [1] of this paper at `http://www.polleres.net/xsparql/`.

As mentioned briefly in the introduction, simple reasoning – which we have not yet incorporated – would significantly improve queries involving RDF data. SPARQL engines that provide (partial) RDFS support could immediately address this point and be

plugged into our implementation. But we plan to go a step further: integrating XSPARQL with Semantic Web Pipes [16] or other SPARQL extensions such as SPARQL++ [19] shall allow more complex intermediate RDF processing than RDFS materialization.

We also plan to apply our results for retrieving metadata from context-aware services and for Semantic Web service communication, respectively, in the EU projects inContext (`http://www.in-context.eu/`) and TripCom (`http://www.tripcom.org/`).

References

1. W. Akthar, J. Kopecký, T. Krennwallner, A. Polleres. XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage. Technical Report DERI-TR-2007-12-14, DERI Galway, Dec. 2007.
2. D. Beckett, B. McBride (eds.). RDF/XML syntax specification (revised). Feb 2004. W3C Rec.
3. D. Beckett. Turtle - Terse RDF Triple Language, Nov. 2007.
4. J. Carroll, P. Stickler. TriX: RDF Triples in XML. Tech. Report HPL-2004-56, HP, May 2004.
5. D. Chamberlin, J. Robie, S. Boag, M. F. Fernández, J. Siméon, D. Florescu (eds.). XQuery 1.0: An XML Query Language. Jan. 2007, W3C Rec.
6. K. Clark, L. Feigenbaum, E. Torres. SPARQL Protocol for RDF, Nov. 2007. W3C Prop. Rec.
7. D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, P. Wadler (eds.). XQuery 1.0 and XPath 2.0 Formal Semantics, Jan. 2007. W3C Rec.
8. M. Droop, M. Flarer, J. Groppe, S. Groppe, V. Linnemann, J. Pinggera, F. Santner, M. Schier, F. Schöpf, H. Staffler, S. Zugal. Translating XPath Queries into SPARQL Queries. *ODBASE2007*.
9. D. Connolly (ed.). Gleaning Resource Descriptions from Dialects of Languages (GRDDL). Sep. 2007, W3C Rec.
10. M. Kay (ed.). XSL Transformations (XSLT) Version 2.0, Jan. 2007. W3C Recommendation.
11. J. Euzenat, P. Shvaiko. *Ontology matching*. Springer, 2007.
12. J. Farrell, H. Lausen (eds.). Semantic Annotations for WSDL and XML Schema. W3C Rec., Aug. 2007.
13. S. Groppe, J. Groppe, V. Linneman, D. Kukulenz, N. Hoeller, and C. Reinke. Embedding SPARQL into XQuery/XSLT. In *SAC2008*, Mar. 2008. To appear.
14. J. Kopecký, T. Vitvar, C. Bournez, J. Farrell. SAWSDL: Semantic Annotations for WSDL and XML Schema. *IEEE Internet Computing*, 11(6):60–67, 2007.
15. A. Malhotra, J. Melton, N. Walsh (eds.). XQuery 1.0 and XPath 2.0 Functions and Operators, Jan. 2007. W3C Rec.
16. C. Morbidoni, A. Polleres, G. Tummarello, D. L. Phuoc. Semantic Web Pipes. Technical Report DERI-TR-2007-11-07, DERI Galway, 11 2007.
17. J. Pérez, M. Arenas, C. Gutierrez. Semantics and Complexity of SPARQL. In *ISWC 2006*.
18. A. Polleres. From SPARQL to Rules (and back). In *Proc. WWW2007*, May 2007.
19. A. Polleres, F. Scharffe, R. Schindlauer. SPARQL++ for mapping between RDF vocabularies. *ODBASE 2007*, Nov. 2007.
20. E. Prud'hommeaux, A. Seaborne (eds.). SPARQL Query Language for RDF, Jan. 2008. W3C Rec.
21. N. Walsh. RDF Twig: Accessing RDF Graphs in XSLT. In *Extreme Markup Languages 2003*.