Using SPARQL with RDFS and OWL Entailment

Birte Glimm

The University of Oxford, Department of Computer Science, UK

Abstract. This chapter accompanies the lecture on SPARQL with entailment regimes at the 7th Reasoning Web Summer School in Galway, Ireland, 2011. SPARQL is a query language and protocol for data specified in the Resource Description Format (RDF). The basic evaluation mechanism for SPARQL queries is based on subgraph matching. The query criteria are given in the form of RDF triples possibly with variables in place of the subject, object, or predicate of a triple, called basic graph patterns. Each instantiation of the variables that yields a subgraph of the queried RDF graph constitutes a solution. The query language further contains capabilities for querying for optional basic graph patterns, alternative graph patterns etc. We first introduce the main features of SPARQL as a query language. In order to define the semantics of a query, we show how a query can be translated to an abstract query, which can then be evaluated according to SPARQL's query evaluation mechanism. Apart from the features of SPARQL 1.0, we also briefly introduce the new features of SPARQL 1.1, which is currently being developed by the Data Access Working Group of the World Wide Web Consortium.

In the second part of these notes, we introduce SPARQL's extension point for basic graph pattern matching. We illustrate how this extension point can be used to define a semantics for basic graph pattern evaluation based on more elaborate semantics such as RDF Schema (RDFS) entailment or OWL entailment. This allows for solutions to a query that implicitly follow from an RDF graph, but which are not necessarily explicitly present. We illustrate what constitutes an extension point and how problems that arise from using a semantic entailment relation can be addressed. We first introduce SPARQL in combination with the RDFS entailment relation and then move on to the more expressive Web Ontology Language OWL. We cover OWL's Direct Semantics, which is based on Description Logics, and the RDF-Based Semantics, which is an extension of the RDFS semantics. For the RDF-Based Semantics we mainly focus on the OWL 2 RL profile, which allows for an efficient implementation using rule engines.

We assume that readers have a basic knowledge of RDF and Turtle, which we use in examples. For the OWL parts, we assume some background in OWL or Description Logics (see lecture notes *Foundations of Description Logics*). The examples for the OWL part are given in Turtle, OWL's functional-style syntax and Description Logics syntax. Although the inferences that are relevant for the example queries are explained, a basic idea about OWL's modeling constructs and their semantics are certainly helpful.

1 Introduction

Query answering is important in the context of the Semantic Web, since it provides a mechanism via which users and applications can interact with ontologies and data. Several query languages have been designed for this purpose, including RDQL, SeRQL and, most recently, SPARQL. We consider the SPARQL [26] query language (pronounce *sparkle*) here, which was standardized in 2008 by the World Wide Web Consortium (W3C) and which is now supported by most RDF triple stores. Currently, the next SPARQL standard is being developed by W3C, named SPARQL 1.1 [13]. Apart from being a query language, the W3C standard also defines a protocol for communicating queries between client and server [11] and a results format for representing query results in XML [5].

The main mechanism for computing query results in SPARQL is subgraph matching: RDF triples in both the queried RDF data and the query pattern are interpreted as nodes and edges of directed graphs, and the resulting query graph is matched to the data graph using variables as wild cards.

In this section, we give some simple examples of SPARQL queries and the query evaluation process. We further introduce the basic ideas behind entailment regimes. In Section 2, we introduce the general features of SPARQL in more detail and explain more formally how a SPARQL query is evaluated. We then introduce SPARQL entailment regimes and explain the design rationals behind the RDFS entailment regime in Section 3. Next, we clarify the relationship between OWL's structural specification and RDF graphs in and introduce the OWL Direct Semantics Entailment Regime in Section 4. Finally, we give some exercises and pointers to additional literature for further reading.

1.1 SPARQL Query Examples

We start with a simple example that illustrates SPARQL's standard query evaluation mechanism, which is based on sub-graph matching. We use Turtle [4] to write down RDF data throughout this chapter and we use the RDF triples shown in Table 1 throughout this and the next section.

Example 1 We consider the following SPARQL query over that data from Table 1:

```
PREFIX  foaf: <http://xmlns.com/foaf/0.1/>
SELECT  ?name?mbox
WHERE  { ?x foaf:name ?name . ?x foaf:mbox ?mbox }
```

For the query, we start by declaring a prefix that allows for abbreviating otherwise long IRIs in the query body. In the remainder we omit this prefix declaration, but assume that the foaf prefix is declared as above. The main part of the query starts with a *select clause* that specifies which variables should be returned with their bindings as part of the result. The *where clause* specifies the conditions that answers have to satisfy. Note that the where clause is still written as a set of triples as in the data above, but the subject and the object are now variables. In order to evaluate such a *basic graph pattern* (BGP), we substitute the variables with terms from the data and if the substitution yields a subgraph of the queried graph, the substituted values are called a solution. The BGP from the above query yields the solutions:

Table 1. Example data used in Section 1 and 2

@prefix	foaf:	<http://xmlns.com/foaf/0.1/>.
_:a	foaf:name	"Birte Glimm".
_:a	foaf:mbox	"b.glimm@googlemail.com".
_:a	foaf:icqChatID	"b.glimm".
_:a	foaf:aimChatID	"b.glimm".
_:b	foaf:name	"Sebastian Rudolph".
_:b	foaf:mbox	<mailto:rudolph@kit.edu>.
_:c	foaf:name	"Pascal Hitzler".
_:c	foaf:aimChatID	"phi".
foaf:icqChatID	rdfs:subPropertyOf	foaf:nick.
foaf:name	rdfs:domain	foaf:Person.

?x	?name	?mbox
_:a	"Birte Glimm"	"b.glimm@googlemail.com"
_:b	"Sebastian Rudolph"	<mailto:rudolph@kit.edu>

Since the select clause only specifies ?name and ?mbox as output variables, a further projection step is required to evaluate the complete query, which only leaves the values for ?name and ?mbox in the query solutions.

1.2 RDF Datasets

Some might be surprised by the absence of a from clause in the query from Example 1, which specifies which data is to be queried. This is because SPARQL queries are executed against an *RDF dataset*, which represents a collection of graphs, and each SPARQL query engine has a default dataset that is normally used. An RDF dataset comprises one graph, called the *default graph*, which does not have a name, and zero or more *named graphs*, where each named graph is identified by an IRI. Unless we change the so called *active graph* for the BGP evaluation with the GRAPH keyword to one of the named graphs, the query is executed against the default graph. In Example 1 the active graph is the default graph, which we have implicitly assumed to contain the given set of triples. Alternatively, a SPARQL query may specify a custom dataset that is to be used for matching by using the FROM and FROM NAMED keyword. In this case, the dataset used for the query consists of a default graph, which is obtained by merging all graphs referred to in a from clause, and a set of (IRI, graph) pairs, one from each from named clause.

Example 2 For an example with custom datasets, let us assume that the data from Table 1 is available under the IRI <http://example.org/foaf/myFoaf>. The following query creates a custom dataset with an empty default graph (no FROM clause) and one named graph.

```
SELECT          ?name?mbox
FROM NAMED <http://example.org/foaf/myFoaf>
WHERE           { GRAPH <http://example.org/foaf/myFoaf>
                      { ?x foaf:name ?name. ?x foaf:mbox ?mbox }
                }
```

Since we used the GRAPH keyword, the active graph for the BGP evaluation is the given named graph. Alternatively, we could use a variable instead of the IRI for the GRAPH keyword, which would evaluate the BGP once over each named graph, binding the graph variable to the corresponding IRI of the named graph. The query answer is the same as for Example 1.

1.3 Blank Nodes in Queries and Query Results

Finally, we want to point out that in our intermediate results for the query from Example 1 we use exactly the same blank node names as in the data from Table 1. This does not have to be the case. Blank nodes just denote the existence of something and we cannot rely on a label being used consistently, it can change when a graph is reloaded or during a merge operation. Furthermore, the query is in fact evaluated against the *scoping graph*, which is equivalent to the active graph, but allows for renaming of blank nodes. Thus, evaluating the BGP could equally result in the solutions where _:a is renamed into _:x and _:b is consistently renamed into _:y. Note, however, that we cannot rename _:a and _:b into the same blank node, nor can we rename the first occurrence of _:b different from the second occurrence.

Since blank node merely denote the existence of something, we can also not understand a blank node in the query as referring to an element with exactly that blank node label in the queried graph. Blank nodes in a BGP can rather be understood as variables, which are immediately projected out after BGP matching, i.e., a blank node cannot occur in the SELECT clause. Since in Example 1 we are not interested in the concrete value that ?x is mapped to, we could equally replace the query pattern with

```
{ _:x foaf:name ?name . _:x foaf:mbox ?mbox }
```

which exactly the same results.

1.4 SPARQL Entailment Regimes

Various W3C standards, including RDF and OWL, provide semantic interpretations for RDF graphs that allow additional RDF statements to be inferred from explicitly given assertions. The entailment regimes in SPARQL 1.1 [12] define how basic graph pattern matching can be defined using semantic entailment relations instead subgraph matching.

Example 3 We again use the data from Table 1 to illustrate the use of inference with the query:

```
SELECT ?name ?nick
WHERE  { ?x foaf:name ?name . ?x foaf:nick ?nick }
```

Using subgraph matching, we do not get an answer. The triple _:a foaf:nick "b.glimm" is, however, entailed by the given triples under RDFS semantics since any subject related with the property foaf:icqChatID is necessarily related to that object also with the property foaf:nick in any RDFS-interpretation that satisfies the data. Under the RDFS entailment regime we expect, therefore, to get "Birte Glimm" as binding for ?name and "b.glimm" as binding for ?nick in the solution.

The entailment regimes developed by the W3C specify exactly what answers we get for several common entailment relations such as RDFS entailment or OWL Direct Semantics entailment. Aspects that have to be addressed include:

- How are the infinitely many axiomatic triples under the RDF(S) semantics handled? These are entailed even by an empty graph and infinite answers, at least due to such axiomatic triples, are rarely desirable.
- How are entailed triples handled that just differ in blank node labels?
- How are inconsistent graphs handled?
- What happens in case of errors?

For OWL's Direct Semantics we further have to address the issue that the semantics is not defined in terms of triples, but in terms of structural objects, which correspond to Description Logic constructs.

1.5 SPARQL as a Protocol

The SPARQL Protocol for RDF defines how a SPARQL query can be conveyed from a query client to a query processor. The protocol is firstly described in an abstract interface independent of any concrete realization, implementation, or binding to another protocol; but a HTTP and SOAP binding of the interface is also provided. We do not further explain the protocol and instead focus on the semantics of SPARQL queries either with subgraph matching (aka simple entailment) or with more elaborate entailment regimes.

2 SPARQL Basics

In the previous examples, we have already seen some basic SPARQL queries. We now make it more precise what parts belong to a query and which choices we have in selecting the data that is returned as the query answer.

2.1 Graph Patterns

The basic selection criteria are specified in the WHERE clause, but before we describe this in more detail, we first recall some basic notions from RDF.

Definition 1. *We write* I *for the set of all* International Resource Identifiers *(IRIs),* L *for the set of all* RDF literals*, and* B *for the set of all* blank nodes*. The set of* RDF terms*, denoted* T*, is* $\mathsf{I} \cup \mathsf{L} \cup \mathsf{B}$.

An RDF graph *is a set of* RDF triples *of the form* (subject, predicate, object) $\in$ $(\mathsf{I} \cup \mathsf{B}) \times \mathsf{I} \times \mathsf{T}$. *We normally omit "RDF" in our terminology if no confusion is likely, and we use Turtle syntax [4] for all examples. The* vocabulary Voc(G) *of a graph* G *is the set of all terms that occur in* G.

Queries are built using a countably infinite set V *of* query variables *disjoint from* T. *A variable* $v \in \mathsf{V}$ *is prefixed by the variable identifier ? or $. The outer-most graph pattern in a query is called the* query pattern.

The variable identifier is not part of the variable name, e.g., $x and ?x denotes the same variable even if both prefixes are used within one query. The variable name itself can contain numbers, letters, and various other admissible symbols [26].

We generally abbreviate IRIs using prefixes rdf, rdfs, owl, xsd, and foaf to refer to the RDF, RDFS, OWL, XML Schema Datatypes, and FOAF namespaces, respectively. We further use the prefix ex for an imaginary example namespace. Prefix declarations for these namespaces are generally omitted in example data and queries.

The simplest form of a WHERE clause consists of a basic graph pattern (BGP), but we can also construct more complex graph patterns by combining smaller patterns in various ways that are described in detail within this section.

Basic Graph Patterns As we have seen in the introductory section, basic graph patterns are the basic building blocks for building a SPARQL query and we can define these formally as follows:

Definition 2. *A* triple pattern *is member of the set* $(\mathsf{T} \cup \mathsf{V}) \times (\mathsf{I} \cup \mathsf{V}) \times (\mathsf{T} \cup \mathsf{V})$*, and a* basic graph pattern *(BGP) is a set of triple patterns.*

According to the above definition, variables can, thus, occur in place of a subject, predicate, or an object. It is also worth pointing out, that the subject of a triple pattern can be a literal although this is not allowed in RDF. This is meant to support (possible future) extensions of RDF. At the moment queries with literals in the subject position can simply not have an answer.

So far we have not seen the effect of blank nodes in BGPs. Since blank nodes do not really refer to a particular resource in the graph, but only denote the existence of something, we cannot expect that the blank node _:a in a BGP is mapped to exactly the blank node _:a in the data as foaf:name in the query is mapped to foaf:name in the queried graph. Instead, blank nodes act similar to variables with the difference that they cannot be selected in the SELECT clause.

Example 4 Since we are not interested in the mappings for ?x in Example 1, we could achieve the same result as with that query pattern:

```
{ _:a foaf:name ?name . _:a foaf:mbox ?mbox }
```

Note that we have deliberately used _:a, which is a blank node label that occurs in the data. The blank node _:a in the BGP acts, however, as a variable and can be substituted any of the blank nodes from the data. Thus, we get the same result with the above query pattern as in Example 1.

As we have seen in Section 1, a WHERE clause that consists of a BGP requires that the set of triple patterns that make up the BGP must all match. In the following, we introduce more complex patters:

– *Group Graph Patterns*, where a set of graph patterns must all match,
– *Optional Graph Patterns*, where additional patterns may extend the solution,
– *Alternative Graph Patterns*, where two or more possible patterns are tried, and
– *Patterns on Named Graphs*, where patterns are matched against named graphs.

Group Graph Patterns Group graph patterns combine patterns conjunctively, similarly to BGPs that combine triple patterns conjunctively. In order to create a group, SPARQL uses curly braces. Grouping by itself is not very useful unless if we only work with basic graph patterns, but it becomes useful when we consider further constructors. For example, we can combine two groups with the UNION keyword, which means that solutions are obtained by matching one or the other group. Before we come to that, we first introduce grouping in more detail.

Example 5 Using groups, patterns of the query from Example 1 can equivalently be written as:

```
{ { ?x foaf:name ?name}
  { }
  { ?x foaf:mbox ?mbox } }
```

We now separated the BGP from Example 1 into three groups. The first and the third group now consists of a single triple pattern. The second group is the *empty* group pattern, which matches to any data. The inclusion or omission of the empty pattern has, therefore, no effect here. A query with only the empty pattern returns always one solution in which any variable that is selected is unbound. Omitting the braces around the two triple patterns as illustrated below:

```
{ ?x foaf:name ?name
  { }
  ?x foaf:mbox ?mbox }
```

leaves us with a group of three elements: a BGP of one triple pattern, an empty group, and again a BGP of one triple pattern.

Alternative Patterns Now that we can group patterns, we can combine groups with other constructors, e.g., the UNION constructor for specifying alternative restrictions. The UNION constructor is a binary operator, i.e., it is used as pattern UNION pattern.

Example 6 We assume that the default graph contains the data from Table 1 and that the SPARQL query is:

```
SELECT ?mbox
WHERE  { { ?x foaf:name "BirteGlimm". ?x foaf:mbox ?mbox }
           UNION
         { ?x foaf:name "SebastianRudolph" . ?x foaf:mbox ?mbox }
       }
```

The result for this query consists of the two email addresses from the queried data. The first email address matches the first BGP and the one for Sebastian Rudolph matches the second BGP, and the results from both BGPs contribute to the final answer due to the UNION keyword.

It is worth noting that the UNION keyword is not denoting an exclusive or and that SPARQL does not have a set semantics as, for example, SQL, so we can have duplicate results in the answer.

Example 7 We illustrate the fact that UNION does not represent an exclusive or and that SPARQL queries can have duplicate results by means of the following query again over the data from Table 1:

```
SELECT ?name ?chatID
WHERE { ?x foaf:name ?name .
        { ?x foaf:icqChatID ?chatID } UNION
        { ?x foaf:aimChatID ?chatID } }
```

The results for the query are as follows:

?name	**?chatID**
"Birte Glimm"	"bgl"
"Birte Glimm"	"bgl"
"Pascal Hitzler"	"phi"

where the first solution results from matching the first alternative and the latter two result from matching the second alternative. The solutions for this query are in fact computed by building the union (without duplicate elimination) from the results of evaluating two graph patters:

```
{ ?x foaf:name ?name.                and    { ?x foaf:name ?name.
  { ?x foaf:icqChatID ?chatID } }             { ?x foaf:aimChatID ?chatID } }
```

In case we use multiple unions, e.g., of the form pattern UNION pattern UNION pattern, this is equivalent to writing: { pattern UNION pattern } UNION pattern, i.e., the UNION operator is left-associative.

Optional Patterns Apart from using the UNION keyword, we can declare some parts as optional using the OPTIONAL keyword, i.e., we allow for only retrieving bindings for these optional parts when these are available.

Example 8 We assume that the default graph contains the data from Table 1 and that the SPARQL query is:

```
SELECT ?name ?mbox
WHERE { ?x foaf:name ?name
        OPTIONAL { ?x foaf:mbox ?mbox } }
```

The result for this query now consists of one additional solution compared to the result for Example 1 in which ?name is bound to "Pascal Hitzler" and ?mbox is unbound. We indicate unbound values by simply leaving the entry in the results table empty:

?name	?mbox
"Birte Glimm"	"b.glimm@googlemail.com"
"Sebastian Rudolph"	<mailto:rudolph@kit.edu>
"Pascal Hitzler"	

The OPTIONAL operator is again binary, i.e., it is used pattern OPTIONAL pattern. Although { OPTIONAL pattern } is syntactically valid, it just abbreviates { {} OPTIONAL pattern }. Like UNION, OPTIONAL is left-associative, i.e., pattern OPTIONAL pattern OPTIONAL pattern is equivalent to: { pattern OPTIONAL pattern } OPTIONAL pattern.

Mixing Optional and Alternative Patterns We can, of course, also mix the use of OPTIONAL and UNION. In this case, left-associativity still applies.

Example 9 The following query over the data from Table 1 gives three results given the left-associativity of UNION and OPTIONAL:

```
SELECT ?name ?chatID ?mbox
WHERE { ?x foaf:name ?name .
        { ?x foaf:icqChatID ?chatID } UNION
        { ?x foaf:aimChatID ?chatID } OPTIONAL
        { ?x foaf:mbox ?mbox } }
```

?name	?chatID	?mbox
"Birte Glimm"	"b.glimm"	"b.glimm@googlemail.com"
"Birte Glimm"	"b.glimm"	"b.glimm@googlemail.com"
"Pascal Hitzler"	"phi"	

After matching the first triple pattern, the union is evaluated. Finally, the optional part is applied to enrich the so far computed solutions. If we were to make the operator preference explicit, we get the following equivalent pattern:

```
{ { ?x foaf:name ?name .
    { { ?x foaf:icqChatID ?chatID } UNION { ?x foaf:aimChatID ?chatID } }
  }
  OPTIONAL { ?x foaf:mbox ?mbox }
}
```

If we do not want the standard left-associate behavior of SPARQL, we have to use braces to enforce a different grouping.

Filters SPARQL *filters* restrict solutions to those for which the filter evaluates to true. The FILTER keyword is followed by a Boolean filter function that evaluates to true or false. Only if the filter function evaluates to true is the solution to be included in the solution sequence.

Example 10 In order to illustrate the use of a filters, we employ the isIRI filter function to filter out results in which the foaf:mbox is not given as an IRI. We use again the data from Table 1 and the query:

```
SELECT ?name ?mbox
WHERE { ?x foaf:name ?name . ?x foaf:mbox ?mbox
            FILTER isIRI(?mbox)
          }
```

Since the isIRI function evaluates to false when ?mbox is bound to the plain literal "b.glimm@googlemail.com", the match cannot be included in the solutions and we get just one result:

?name	**?mbox**
"Sebastian Rudolph"	<mailto:rudolph@kit.edu>

There are quite a range of filter functions, e.g., functions for comparing numerical values or date, filtering strings according to a regular expression, test whether a binding is a blank node, or whether a variable is bound at all. For more details, we refer to the SPARQL Query specification [26].

Literals The general syntax for literals in a SPARQL query is a string enclosed in either double quotes ("...") or single quotes ('...') with either an optional language tag (introduced by @) or an optional datatype IRI or prefixed name (introduced by ^^).

For convenience, integers can be written without quotation marks and an explicit datatype IRI. Such literals are interpreted as typed literals of datatype xsd:integer, xsd:decimal if there is no '.' in the number; otherwise the number is interpreted as xsd:decimal if no exponent is given and as xsd:double otherwise. Literals of type xsd:boolean can also be written as `true` or `false`.

To facilitate writing literal values which themselves contain quotation marks or which are long and contain newline characters, SPARQL provides an additional quoting construct in which literals are enclosed in three single- or double-quotation marks.

Examples of literal syntax in SPARQL include:

- "chat" or 'chat'
- "chat"@fr with language tag "fr"
- "xyz"^^<http://example.org/ns/userDatatype>
- "abc"^^appNS:appDataType
- '''The librarian said, "Perhaps you would enjoy 'War and Peace'." '''
- 1, which is the same as "1"^^xsd:integer
- 1.3, which is the same as "1.3"^^xsd:decimal
- 1.300, which is the same as "1.300"^^xsd:decimal
- 1.0e6, which is the same as "1.0e6"^^xsd:double
- true, which is the same as "true"^^xsd:boolean
- false, which is the same as "false"^^xsd:boolean

2.2 Result Formats

SPARQL has four query forms. These query forms use the solutions from pattern matching to form result sets or RDF graphs. The query forms are:

- SELECT returns all, or a subset of, the variables bound in a query pattern match.
- CONSTRUCT returns an RDF graph constructed by substituting variables in a set of triple templates.
- ASK returns a boolean indicating whether a query pattern matches or not.
- DESCRIBE returns an RDF graph that describes the resources found.

The XML results format of SPARQL can be used to serialize the result set from a select query or the boolean result of an ask query.

SELECT queries return variables and their bindings directly. The syntax SELECT * is an abbreviation that selects all of the variables in a query that are in scope. The definition of scope becomes relevant when making use of the SPARQL 1.1 sub-select feature in which case only variables that are projected in the sub-query are visible in the enclosing query.

CONSTRUCT queries return a single RDF graph. The result is an RDF graph formed by taking the specified graph template and by instantiating it with each query solution in the solution sequence. The triples obtained from each solution are combining into a single RDF graph by set union.

If any such instantiation produces a triple containing an unbound variable or an illegal RDF construct, e.g., containing a literal in subject or predicate position, then that triple is not included in the output RDF graph. The graph template can contain triples with no variables (known as ground or explicit triples), and these also appear in the output RDF graph returned by the CONSTRUCT query form.

A template can create an RDF graph containing blank nodes. The blank node labels are scoped to the template for each solution. If the same label occurs twice in a template, then there will be one blank node created for each query solution, but there will be different blank nodes for triples generated by different query solutions.

Example 11 In order to see an example for construct and the effect of blank nodes in the template, we se the following query over the data from Table 1:

```
CONSTRUCT { ?x rdf:type foaf:Person . ?x foaf:givenName _:x }
WHERE     { ?x foaf:name ?name }
```

Evaluating the query pattern yields the following bindings:

?x	?name
_:a	"Birte Glimm"
_:b	"Sebastian Rudolph"
_:c	"Pascal Hitzler"

Instantiating and building the set union of the template then results in:

```
_:u   rdf:type          foaf:Person.
_:u   foaf:givenName    _:x1
_:v   rdf:type          foaf:Person.
_:v   foaf:givenName    _:x2
_:w   rdf:type          foaf:Person.
_:w   foaf:givenName    _:x3
```

Note that the variable binds to blank nodes in the data and there is not even a guarantee that in the intermediate results the same blank node labels are used. In the constructed data a different blank node label is created. Similarly, the blank node in the template just causes a different blank node label to be created each time the template is instantiated.

ASK queries test whether or not a query pattern has a solution. No information is returned about the possible query solutions, just whether or not a solution exists.

Example 12 The following queries illustrates the use of the ASK query form for the data from Table 1:

```
ASK { ?x foaf:name "Birte Glimm" }
```

The result to this query is `true` or `yes` since there is a possible binding for ?x in the data.

DESCRIBE queries return a single RDF graph containing RDF data about resources.

Example 13 Possible examples for DESCRIBE are the following:

```
DESCRIBE foaf:Person   or
DESCRIBE ?x WHERE { ?x foaf:name "Birte Glimm" }
```

The exact output is not prescribed by the SPARQL Query specification, i.e., results depend on the SPARQL query processor and can vary between systems. The resulting RDF graph can be complex and can, for example, mention other resources that are somehow related to the given resource. For example, whether a property denotes an inverse or an inverse functional property, or the name and mbox if the resource that is to be described is from a FOAF file.

2.3 Solution Modifiers

Query patterns generate a multiset of solutions, each solution being a partial function from variables to RDF terms. These solutions are then treated as a sequence (a solution sequence), initially in no specific order; any sequence modifiers are then applied to create another sequence. Finally, the sequence is used to generate one of the results of a SPARQL query form. A *solution sequence modifier* is one of:

– *Order modifier* to put the solutions in order

- *Projection modifier* to choose certain variables and eliminate others from the solutions
- *Distinct modifier* to eliminate duplicate solutions
- *Reduced modifier* to permit elimination of some non-unique solutions
- *Offset modifier* to control where the solutions start from in the overall sequence of solutions
- *Limit modifier* to restrict the number of solutions

ORDER BY is a keyword that establishes an order within a solution sequence. It is followed by a sequence of order comparators, composed of an expression and an optional order modifier (either ASC(·) or DESC(·)). Each ordering comparator is either ascending (indicated by the ASC(·) modifier or by no modifier) or descending (indicated by the DESC(·) modifier).

Example 14 We illustrate the use ORDER BY with the following query over the data from Table 1:

```
SELECT     ?name
WHERE      { ?x foaf:name ?name }
ORDER BY ?name
```

Since ascending is the default ordering, the query is equivalent to:

```
SELECT     ?name
WHERE      { ?x foaf:name ?name }
ORDER BY ASC(?name)
```

The results are now ordered according to the bindings for the variable ?name:

?name
"Birte Glimm"
"Pascal Hitzler"
"Sebastian Rudolph"

With the DESC(·) keyword we would get the exact opposite order.

The ascending order of two solutions with respect to an ordering comparator is established by substituting the solution bindings into the expressions and comparing them with the < operator, which is defined by the SPARQL Query specification for numerics, simple literals, xsd:string, xsd:boolean, and xsd:dateTime. Descending order is the reverse of the ascending order. Ordering never changes the cardinality of solutions.

Pairs of IRIs are ordered by comparing them as simple literals. SPARQL also fixes an order between some kinds of RDF terms that would not otherwise be ordered (given here from lowest in the order):

1. no value assigned to the variable or expression in this solution,
2. blank nodes,
3. IRIs,
4. RDF literals.

A plain literal is lower than an RDF literal with type xsd:string of the same lexical form. Note that SPARQL does not define a total ordering over all possible RDF terms; a few examples of pairs of terms for which the relative order is undefined are:

- a simple literal and a literal with a language tag, e.g., "a" and "a"en_gb,
- two literals with language tags, e.g., "a"en_gb and "b"en_gb,
- a simple literal and an xsd:string, e.g., "a" and "a"^^xsd:string,
- a simple literal and a literal with a supported data type, e.g., "a" and "1"^^xsd:integer,
- two unsupported data types, e.g., "1"^^my:integer and "2"^^my:integer
- a supported data type and an unsupported data type, e.g., "1"^^xsd:integer and "2"^^my:integer.

The ORDER BY clause can also contain a LIMIT n and an OFFSET m condition to limit the number of returned results to *n* and to start only with the m^{th} result. Using LIMIT and OFFSET to select different subsets of the query solutions will not be useful unless the order is made predictable by using ORDER BY.

Using ORDER BY on a solution sequence for a CONSTRUCT or DESCRIBE query has no direct effect because only SELECT returns a sequence of results. Used in combination with LIMIT and OFFSET, ORDER BY can be used to return results generated from a different slice of the solution sequence. An ASK query does not include ORDER BY, LIMIT or OFFSET.

Projection can be used to transform the solution sequence into one involving only a subset of the variables. For each solution in the sequence, a new solution is formed using a specified selection of the variables as specified in the SELECT clause.

We have used projection already any many previous examples, so do not give another example here.

DISTINCT is a solution modifier that causes the elimination of duplicate solutions. Specifically, each solution that binds the same variables to the same RDF terms as another solution is eliminated from the solution set.

REDUCED is not as strong as DISTINCT because it permits duplicate elimination, but does not enforce it. Thus, some duplicates might be eliminated, whereas other remain in the solution sequence.

2.4 SPARQL Algebra Processing

So far we have mainly used examples to illustrate the effect of different SPARQL operators. In this section, we precisely define how a the result for a SPARQL query is computed, which requires transforming a query string into a SPARQL algebra object that is then evaluated in order to compute the query result.

The first step towards obtaining an algebra object for a query string is parsing. In the parsing process, we expanding abbreviations for IRIs and triple patterns, e.g., for triple patterns that use Turtle's comma or semicolon abbreviations.

Table 2. SPARQL 1.0 grammar elements that make up a query pattern

```
GroupGraphPattern         ::= '{' TriplesBlock?
                                ( ( GraphPatternNotTriples | Filter ) '.'?
                                  TriplesBlock? )*
                              '}'
GraphPatternNotTriples    ::= OptionalGraphPattern | GroupOrUnionGraphPattern
                              | GraphGraphPattern
OptionalGraphPattern      ::= 'OPTIONAL' GroupGraphPattern
GraphGraphPattern         ::= 'GRAPH' VarOrIRIref GroupGraphPattern
GroupOrUnionGraphPattern  ::= GroupGraphPattern ( 'UNION' GroupGraphPattern )*
Filter                    ::= 'FILTER' Constraint
```

Translating SPARQL Query Patterns to Algebra Expressions Parsing of a SPARQL query string involves expanding abbreviations for IRIs and triple patterns and the construction of an abstract syntax tree that can then be transformed into a SPARQL algebra expressions. The semantics of a query is then given based on the algebra expression. After parsing, we have an abstract syntax tree that represents the expanded query string and which is then converted into an algebra object.

Example 15 The algebra expression corresponding to the simple query

SELECT ?s WHERE { ?s :p ?o }

is Project(Bgp(?s <http://example.org#p> ?o), {?s}) if we assume that the empty prefix is defined as <http://example.org#>. The algebra expression is evaluated inside out, i.e., we first evaluate the pattern ?s <http://example.org#p> ?o given as parameter to the Bgp algebra expression; then a projection is performed so that only the values for the variable ?s remain.

We start by looking into the translation of a query pattern before we come to solution modifiers and other parts that are not related to the query pattern. We restrict our explanation to SPARQL 1.0 elements; SPARQL 1.1 works quite similar, but due to the much increased features the translation gets far more involved.

The translation is defined in terms of objects from the SPARQL grammar, i.e., one has to understand which part of a query pattern has been produced by a certain grammar object. The grammar defines a query pattern as a `GroupGraphPattern` and we give the relevant grammar part in Table 2, where `TriplesBlock` denotes a basic graph pattern, `VarOrIRIref` denotes a variable or an IRI, and `Constraint` represents a filter expression.

We first define a function algbr that takes a query pattern and inductively translates it into a SPARQL algebra expression. The algebra objects that we encounter in the translation process are

– Bgp, for expressing that a BGP has to be evaluated, e.g., by performing subgraph matching,
– Join, for joining results, e.g., from different groups,
– LeftJoin, for combining results with optional values,

– Filter, for filtering results according to a filter expression,
– Union, for combining results from alternatives,
– Graph, for evaluating a query part on a named graph.

We further encounter the empty pattern, denoted Z, which is a basic graph pattern that evaluates to an empty solution mapping, i.e., to a solution mapping that does not map any variable to a value. Thus, the empty pattern can be joined with any other pattern without any effect, i.e., Z is the identity for join.

Definition 3. *We define the function* algbr(P) *as follows: If* P *is* `TriplesBlock`*, then*

$$\mathsf{algbr(P)} := \begin{cases} Z & \textit{if } \mathsf{P} \textit{ is empty} \\ \mathsf{Bgp(P)} & \textit{otherwise.} \end{cases}$$

If P *is* `GroupOrUnionGraphPattern` *with elements* $\mathsf{e_1, \dots, e_n}$*, then*

$$\mathsf{algbr(P)} := \begin{cases} \mathsf{algbr(e_1)} & \textit{for } n = 1 \textit{ and} \\ \mathsf{Union(algbr(e_1), algbr(e_2\ UNION \dots UNION\ e_n))} & \textit{otherwise.} \end{cases}$$

If P *is* `GraphGraphPattern` *of the form* term GRAPH P′*, then*

$$\mathsf{algbr(P)} := \mathsf{Graph(term, algbr(P'))}.$$

If P *is* `GroupGraphPattern` *containing filter elements* $\mathsf{f_1, \dots, f_n}$ *and other elements* $\mathsf{e_1, \dots, e_m}$ *then*

$$\mathsf{algbr(P)} := \begin{cases} \mathsf{Filter(f_1 \&\& \dots \&\& f_n, translateGroup(e_1, \dots, e_m))} & \textit{if } n > 0 \textit{ and} \\ \mathsf{translateGroup(e_1, \dots, e_m)} & \textit{otherwise,} \end{cases}$$

where translateGroup *is described in Algorithm 1 and && is SPARQL's conjunction operator for filter expressions.*

The resulting algebra objects can be *simplified* by exploiting the join identity property of the empty pattern Z: we can replace Join(Z, A) by A and Join(A, Z) by A.

Example 16 For example, the simple query pattern
{ ?s ?p ?o } is translated to Join(Z, Bgp(?s ?p ?o))
since { ?s ?p ?o } is an instance of `GroupGraphPattern` (as every query pattern), which is translated according to Algorithm 1 (line 10), where the triple pattern itself is translated as `TriplesBlock`. According to the join identity simplification, the expression can be simplified to Bgp(?s ?p ?o).

In order to see some examples of the translation, we go through several of the example query patterns that we have encountered so far.

Algorithm 1 Translation of non-filter elements in group graph patterns

```
Algorithm: translateGroup($e_1, \ldots, e_n$)
Input: $e_1, \ldots, e_n$: the list of non-filter elements in a group pattern
Output: a SPARQL algebra expression A
 1: A := Z {the empty pattern}
 2: for $i = 1, \ldots, n$ do
 3:   if $e_i$ is of the form OPTIONAL pattern then
 4:     if algbr(pattern) is of the form Filter(F, A′) then
 5:       A := LeftJoin(A, A′, F)
 6:     else
 7:       A := LeftJoin(A, algbr(pattern), true)
 8:     end if
 9:   else
10:     A := Join(A, algbr($e_i$))
11:   end if
12: end for
13: return A
```

Example 1:
Query pattern: Join(Z, Bgp(?x foaf:name ?name. ?x foaf:mbox ?mbox)),
Simplification: Bgp(?x foaf:name ?name. ?x foaf:mbox ?mbox).

Example 2:
Query pattern: Join(Z, Graph(*iri*, Join(Z, Bgp(BGP_1)))),
Simplification: Graph(*iri*, Bgp(BGP_1)).
We use iri *and* BGP_1 *instead of the given graph IRI and BGP.*

Example 3:
Query pattern: Join(Z, Bgp(?x foaf:name ?name. ?x foaf:nick ?nick)),
Simplification: Bgp(?x foaf:name ?name. ?x foaf:nick ?nick).
Note that the example was used to illustrate the effects of entailment regimes, but this does not influence the conversion to algebra objects. The only effect is that the Bgp(·) *algebra objects are evaluated differently.*

Example 5:
Query pattern: Join(Join(Join(Z, Bgp(TP_1)), Z), Bgp(TP_2)),
Simplification: Join(Bgp(?x foaf:name ?name), Bgp(?x foaf:mbox ?mbox)).
We abbreviate the two triples patterns with TP_1 *and* TP_2*, respectively. Although the query from Example 5 yields the same results as the query from Example 1 on any data, its algebra version is different due to the use of groups. We first used Algorithm 1 on the three elements of the query pattern, which is a group graph pattern. Each element itself is then translated as a* `GroupOrUnionGraphPattern` *using the case for single elements. A query optimizer might further modify this algebra object so that it becomes the same as the simplified version of the algebra expression for Example 1.*

Example 6:
Query pattern: Join(Z, Union(Bgp(BGP_1), Bgp(BGP_2))),
Simplification: Union(Bgp(BGP_1), Bgp(BGP_2)).
We abbreviate the two BGPs from the union pattern with BGP_1 *and* BGP_2*, respectively.*

Example 7:
Query pattern: Join(Join(Z, Bgp(tp_1)), Union(Bgp(tp_2), Bgp(tp_3))),

Simplification: $\mathsf{Join}(\mathsf{Bgp}(\mathsf{tp}_1), \mathsf{Union}(\mathsf{Bgp}(\mathsf{tp}_2), \mathsf{Bgp}(\mathsf{tp}_3)))$.

We abbreviate the three triple patterns from the example with tp_1 *to* tp_3*, respectively.*

Example 8:

Query pattern: $\mathsf{LeftJoin}(\mathsf{Join}(\mathsf{Z}, \mathsf{Bgp}(\mathsf{tp}_1)), \mathsf{Bgp}(\mathsf{tp}_2), \mathsf{true})$,

Simplification: $\mathsf{LeftJoin}(\mathsf{Bgp}(\mathsf{tp}_1), \mathsf{Bgp}(\mathsf{tp}_2), \mathsf{true})$.

The triple patterns from the example are abbreviated with tp_1 *and* tp_2*, respectively.*

Example 9:

Query pattern:
$\mathsf{LeftJoin}(\mathsf{Join}(\mathsf{Join}(\mathsf{Z}, \mathsf{Bgp}(\mathsf{tp}_1)), \mathsf{Union}(\mathsf{Bgp}(\mathsf{tp}_2), \mathsf{Bgp}(\mathsf{tp}_3))), \mathsf{Bgp}(\mathsf{tp}_4), \mathsf{true})$,

Simplification:
$\mathsf{LeftJoin}(\mathsf{Join}(\mathsf{Bgp}(\mathsf{tp}_1), \mathsf{Union}(\mathsf{Bgp}(\mathsf{tp}_2), \mathsf{Bgp}(\mathsf{tp}_3))), \mathsf{Bgp}(\mathsf{tp}_4), \mathsf{true})$.

We again abbreviate the triple patterns from the example with tp_1 *to* tp_4*, respectively.*

Example 10:

Query pattern: $\mathsf{Filter}(\mathit{isIRI}(?\mathsf{mbox}), \mathsf{Join}(\mathsf{Z}, \mathsf{Bgp}(\mathsf{bgp}_1)))$,

Simplification: $\mathsf{Filter}(\mathit{isIRI}(?\mathsf{mbox}), \mathsf{Bgp}(\mathsf{bgp}_1))$.

For the translation of the query pattern we use the first case of translating group graph patterns. We abbreviate the basic graph pattern from the example with bgp_1.

We omit the examples for NOT EXISTS, EXISTS, and MINUS since these are SPARQL 1.1 features, for which we do not go into details in the algebra translation.

Evaluating Algebra Expressions for Query Patterns In order to define the evaluation of an algebra object for a query pattern, we first define the most basic operation, i.e., the evaluation of a basic graph pattern.

Definition 4. *Evaluating a SPARQL graph pattern results in a* solution sequence *that lists possible bindings of query variables to RDF terms in the active graph. Such bindings are represented by partial functions* μ *from* V *to* T*, called* solution mappings*. For a solution mapping* μ *– and more generally for any (partial) function – the set of elements on which* μ *is defined is the* domain $\mathsf{dom}(\mu)$ *of* μ*, and the set* $\mathsf{ran}(\mu) := \{\mu(x) \mid x \in \mathsf{dom}(\mu)\}$ *is the* range *of* μ*. For a graph pattern* GP*, we use* $\mu(\mathsf{GP})$ *to denote the pattern obtained by applying* μ *to all elements of* GP *in* $\mathsf{dom}(\mu)$*.*

This convention is extended in the obvious way to filter expressions, and to all functions that are defined on variables or terms.

The order of solution sequences is relevant for later processing steps in SPARQL, but not for obtaining the solutions for a graph pattern. Thus, we obtain a *solution multiset* when evaluating a basic graph pattern, or, more generally, any SPARQL graph patterns.

Definition 5. *A* multiset *over an* underlying set S *is a total function* $M \colon S \to \mathbf{N}^+ \cup \{\omega\}$ *where* $\mathbf{N}^+$ *are the positive natural numbers, and* $\omega > n$ *for all* $n \in \mathbf{N}^+$*. The value* $M(s)$ *is the* multiplicity *of* $s \in S$*, and* ω *denotes a countably infinite number of occurrences.*

Infinitely many occurrences of individual solution mappings are indeed possible when considering SPARQL entailment regimes, although a major concern when defining extensions to basic graph pattern matching is how to avoid sources of infinite solutions.

Table 3. Evaluation of algebraic operators for query patterns in SPARQL over a dataset D, where the multiplicity functions M, M_1, and M_2 are assumed to be those for the multisets $[\![\mathsf{GP}]\!]_{\mathsf{D,G}}$, $[\![\mathsf{GP_1}]\!]_{\mathsf{D,G}}$, and $[\![\mathsf{GP_2}]\!]_{\mathsf{D,G}}$, v is a variable, and iri is an IRI

$$
\begin{aligned}
[\![\mathsf{Union}(\mathsf{GP_1},\mathsf{GP_2})]\!]_{\mathsf{D,G}} &:= \big\{(\mu,n) \mid n = M_1(\mu) + M_2(\mu) > 0\big\} \\
[\![\mathsf{Join}(\mathsf{GP_1},\mathsf{GP_2})]\!]_{\mathsf{D,G}} &:= \big\{(\mu,n) \mid n = \textstyle\sum_{(\mu_1,\mu_2)\in J(\mu)} (M_1(\mu_1) * M_2(\mu_2)) > 0\big\} \text{ where} \\
&\quad\; J(\mu) := \{(\mu_1,\mu_2) \mid \mu_1,\mu_2 \text{ compatible and } \mu = \mu_1 \cup \mu_2\} \\
[\![\mathsf{Filter}(\mathsf{F},\mathsf{GP})]\!]_{\mathsf{D,G}} &:= \big\{(\mu,n) \mid M(\mu) = n > 0 \text{ and } [\![\mu(\mathsf{F})]\!] = \mathsf{true}\big\} \\
[\![\mathsf{LeftJoin}(\mathsf{GP_1},\mathsf{GP_2},\mathsf{F})]\!]_{\mathsf{D,G}} &:= [\![\mathsf{Filter}(\mathsf{F},\mathsf{Join}(\mathsf{GP_1},\mathsf{GP_2}))]\!]_{\mathsf{D,G}} \;\cup \\
&\quad\; \big\{(\mu_1, M_1(\mu_1)) \mid \text{for all } \mu_2 \text{ with } M_2(\mu_2) > 0 : \mu_1 \text{ and } \mu_2 \text{ are} \\
&\qquad\qquad \text{incompatible or } [\![(\mu_1 \cup \mu_2)(\mathsf{F})]\!] = \mathsf{false}\big\} \\
[\![\mathsf{Graph}(\mathsf{iri},\mathsf{GP})]\!]_{\mathsf{D,G}} &:= \begin{cases} [\![\mathsf{GP}]\!]_{\mathsf{D,G_{iri}}} & \text{if iri is an IRI with } (\mathsf{iri},\mathsf{G_{iri}}) \in \mathsf{D} \\ \{\!\!\{\}\!\!\} & \text{otherwise} \end{cases} \\
[\![\mathsf{Graph}(\mathsf{v},\mathsf{GP})]\!]_{\mathsf{D,G}} &:= [\![\mathsf{Union}(\mathsf{Join}(\mathsf{Graph}(\mathsf{iri_n},\mathsf{GP}), \{\!\!\{\mu\colon \mathsf{v} \mapsto \mathsf{iri_n}\}\!\!\}), \\
&\qquad \mathsf{Union}(\mathsf{Join}(\mathsf{Graph}(\mathsf{iri_{n-1}},\mathsf{GP}), \{\!\!\{\mu\colon \mathsf{v} \mapsto \mathsf{iri_{n-1}}\}\!\!\}), \\
&\qquad\quad \mathsf{Union}(\ldots, \\
&\qquad\qquad \mathsf{Join}(\mathsf{Graph}(\mathsf{iri_1},\mathsf{GP}), \{\!\!\{\mu\colon \mathsf{v} \mapsto \mathsf{iri_1}\}\!\!\}))))]\!]_{\mathsf{D,G}} \\
&\quad\; \text{for } \mathsf{iri_1},\ldots,\mathsf{iri_n} \text{ the IRIs of the named graphs in } \mathsf{D}
\end{aligned}
$$

We often represent a multiset M with underlying set S by the set $\{(s, M(s)) \mid s \in S\}$. Accordingly, we may write $(s,n) \in M$ if $M(s) = n$. Also, we assume that $M(s)$ denotes 0 whenever $s \notin S$. In some cases, it is also convenient to use a set-like notation where repeated elements are allowed, e.g. writing $\{\!\!\{a,b,b\}\!\!\}$ for the multiset M with underlying set $\{a,b\}$, $M(a) = 1$, and $M(b) = 2$.

To define the solution multiset for a BGP under the simple semantics, we still need to consider the effect of blank nodes. Intuitively, these act like variables that are projected out of a query result, and thus they may lead to duplicate solution mappings. This is accounted for using RDF instance mappings as follows:

Definition 6. *An* RDF instance mapping *is a partial function* $\sigma\colon \mathsf{B} \to \mathsf{T}$ *from blank nodes to RDF terms. We extend* σ *to pattern graphs and filters as done for solution mappings above. The* solution multiset $[\![\mathsf{BGP}]\!]_{\mathsf{D,G}}$ *for a basic graph pattern* BGP *over the dataset* D *with active graph* G *is the following multiset of solution mappings:*

$$
\begin{aligned}
\{(\mu,n) \;\mid\; & \mathsf{dom}(\mu) = \mathsf{V}(\mathsf{BGP}), \textit{ and } n \textit{ is the maximal number such that} \\
& \sigma_1,\ldots,\sigma_n \textit{ are distinct RDF instance mappings such that, for all } 1 \le i \le n, \\
& \mathsf{dom}(\sigma_i) = \mathsf{B}(\mathsf{BGP}) \textit{ and } \mu(\sigma_i(\mathsf{BGP})) \textit{ is a subgraph of } \mathsf{G}\}.
\end{aligned}
$$

Note that the number n in the definition of $[\![\mathsf{BGP}]\!]_{\mathsf{D,G}}$ is always finite.

The algebraic operators that are required for evaluating non-basic graph patterns correspond to operations on multisets of solution mappings. This remains unchanged even if we use an entailment regime different from SPARQL's standard simple entailment. To take infinite multiplicities into account, which can occur in some entailment regimes, we assume $\omega + n = n + \omega = \omega$ for all $n \ge 0$, $\omega * n = n * \omega = \omega$ for all $n > 0$ and $\omega * 0 = 0 * \omega = 0$. We denote the truth value from evaluating a filter F by $[\![\mathsf{F}]\!]$.

Definition 7. *Two solution mappings μ_1 and μ_2 are* compatible *if $\mu_1(v) = \mu_2(v)$ for all $v \in \mathsf{dom}(\mu_1) \cap \mathsf{dom}(\mu_2)$. If this is the case, a solution mapping $\mu_1 \cup \mu_2$ is defined by setting $(\mu_1 \cup \mu_2)(v) := \mu_1(v)$ if $v \in \mathsf{dom}(\mu_1)$, and $(\mu_1 \cup \mu_2)(v) := \mu_2(v)$ otherwise.*

The evaluation *of a graph pattern over* G*, denoted $[\![\cdot]\!]_{\mathsf{D,G}}$, is defined as in Table 3, where the multiplicity functions M / M_1 / M_2 are assumed to be those for the multisets* $[\![\mathsf{GP}]\!]_{\mathsf{D,G}}$ / $[\![\mathsf{GP_1}]\!]_{\mathsf{D,G}}$ / $[\![\mathsf{GP_2}]\!]_{\mathsf{D,G}}$.

Note that, for brevity, we join an algebra object with a multiset in the evaluation of Graph(v, GP) with v a variable, which is strictly speaking not possible since both of the joined elements should be algebra objects that are then evaluated in the join evaluation.

Translating and Evaluating SPARQL Queries Apart from the query pattern itself, the solution modifiers of a query are also translated into corresponding algebra objects. The resulting algebra object together with a dataset for the query and a query form defines a SPARQL abstract query.

Definition 8. *Given a SPARQL query* Q *with query pattern* P*, we step by step construct an algebra expression* E *as follows:*

1. E := ToList(algbr(P))*, where* ToList *turns a multiset into a sequence with the same elements and cardinality. There is no implied ordering to the sequence; duplicates need not be adjacent.*
2. E := OrderBy(E, $(\mathsf{c_1}, \ldots, \mathsf{c_n})$) *if the query string has an* ORDER BY *clause, where $c_1, \ldots, c_n$ the order comparators in Q.*
3. E := Project(E, vars) *if the query form is* SELECT*, where* vars *is the set of variables mentioned in the* SELECT *clause or all named variables in the query if* SELECT * *is used.*[1]
4. E := Distinct(E) *if the query form is* SELECT *and the query contains the* DISTINCT *keyword.*
5. E := Reduced(E) *if the query form is* SELECT *and the query contains the* REDUCED *keyword.*
6. E := Slice(E, start, length) *if the query contains* OFFSET start *or* LIMIT length*, where* start *defaults to* 0 *and* length *defaults to* (size(E) − start) *with* size(E) *denoting the cardinality of E.*
7. E := Construct(E, templ) *if the query form is* CONSTRUCT *and* templ *is the template of the query.*
8. E := Describe(E, VarsRes) *if the query form is* DESCRIBE*, where* VarsRes *is the set of variables and resources mentioned in the* DESCRIBE *clause or all named variables in the query if* DESCRIBE * *is used.*

The algebra expression for Q*, denoted* algbr(Q)*, is* E.

We define the RDF dataset for Q *as follows: if* Q *contains a* FROM *or* FROM NAMED *clause, then the* RDF dataset D for Q *is* $\{\mathsf{G}, (\mathsf{iri_1}, \mathsf{G_1}), \ldots, (\mathsf{iri_n}, \mathsf{G_n})\}$ *where the default*

[1] Note that for SPARQL 1.1, * only refers to variables that are in scope, e.g., if the query contains a sub-query, then only variables that are selected in the sub-query are in scope for the enclosing query.

Table 4. Evaluation of algebraic operators for queries over a dataset D with default graph G

$$
\begin{aligned}
[\![\mathsf{ToList(E)}]\!]_{\mathsf{D,G}} &:= (\mu_1,\ldots,\mu_n) \text{ for } \{\!\{\mu_1,\ldots,\mu_n\}\!\} = [\![\mathsf{E}]\!]_{\mathsf{D,G}}\\
[\![\mathsf{OrderBy}(\mathsf{E},(\mathsf{c}_1,\ldots,\mathsf{c}_\mathsf{m}))]\!]_{\mathsf{D,G}} &:= (\mu_1,\ldots,\mu_n) \text{ such that } (\mu_1,\ldots,\mu_n) \text{ satisfies } (c_1,\ldots,c_m)\\
&\quad\;\, \text{and } \{\!\{\mu_1,\ldots,\mu_n\}\!\} = \{\!\{\mu \mid \mu \in [\![\mathsf{E}]\!]_{\mathsf{D,G}}\}\!\}\\
[\![\mathsf{Project}(\mathsf{E},\mathsf{vars})]\!]_{\mathsf{D,G}} &:= (\mu'_1,\ldots,\mu'_n) \text{ with } (\mu_1,\ldots,\mu_n) = [\![\mathsf{E}]\!]_{\mathsf{D,G}}, \mathsf{dom}(\mu'_\mathsf{i}) = \mathsf{vars} \subseteq \mathsf{dom}(\mu_\mathsf{i}),\\
&\quad\;\, \text{and } \mu'_i \text{ is compatible with } \mu_i \text{ for } 1 \le i \le n\\
[\![\mathsf{Distinct(E)}]\!]_{\mathsf{D,G}} &:= (\mu_1,\ldots,\mu_m) \text{ with } \{\!\{\mu_1,\ldots,\mu_n\}\!\} = \{\!\{\mu \mid \mu \in \{\mu \mid \mu \in [\![\mathsf{E}]\!]_{\mathsf{D,G}}\}\}\!\}\\
&\quad\;\, \text{and } (\mu_1,\ldots,\mu_m) \text{ preserves the order of } [\![\mathsf{E}]\!]_{\mathsf{D,G}}\\
[\![\mathsf{Reduced(E)}]\!]_{\mathsf{D,G}} &:= (\mu_1,\ldots,\mu_m) \text{ with } \{\!\{\mu_1,\ldots,\mu_n\}\!\} \subseteq \{\!\{\mu \mid \mu \in [\![\mathsf{E}]\!]_{\mathsf{D,G}}\}\!\},\\
&\quad\;\, \{\mu_1,\ldots,\mu_m\} = \{\mu \mid \mu \in [\![\mathsf{E}]\!]_{\mathsf{D,G}}\}\\
&\quad\;\, \text{and } (\mu_1,\ldots,\mu_m) \text{ preserves the order of } [\![\mathsf{E}]\!]_{\mathsf{D,G}}\\
[\![\mathsf{Slice}(\mathsf{E},\mathsf{start},\mathsf{length})]\!]_{\mathsf{D,G}} &:= (\mu_{start},\ldots,\mu_{start+length}) \text{ for } (\mu_1,\ldots,\mu_m) = [\![\mathsf{E}]\!]_{\mathsf{D,G}}\\
[\![\mathsf{Construct}(\mathsf{E},\mathsf{templ})]\!]_{\mathsf{D,G}} &:= \{\mu_i(\mathsf{templ_i}) \mid \{\!\{\mu_1,\ldots,\mu_n\}\!\} = [\![\mathsf{E}]\!]_{\mathsf{D,G}}, 1 \le i \le n, \mathsf{templ_i} \text{ is graph}\\
&\quad\;\, \text{equivalent to } \mathsf{templ}, \mu_i(\mathsf{templ_i}) \text{ is valid RDF, and}\\
&\quad\;\, \mathsf{B}(\mathsf{templ_i}) \cap \textstyle\bigcup_{1\le j\le n, j\ne i}(\mathsf{B}(\mathsf{templ_j}) \cup \mathsf{ran}(\mathsf{mu_i})) = \emptyset\}\\
[\![\mathsf{Describe}(\mathsf{E},\mathsf{VarsRes})]\!]_{\mathsf{D,G}} &:= \{\mathsf{desc}(\mu_i(\mathsf{VarsRes})) \mid \{\!\{\mu_1,\ldots,\mu_n\}\!\} = [\![\mathsf{E}]\!]_{\mathsf{D,G}}, 1 \le i \le n\\
&\quad\;\, \text{where } \mathsf{descr} \text{ generates a system-dependent description}\\
&\quad\;\, \text{for the given resources}
\end{aligned}
$$

graph G *is the RDF merge of the graphs referred to in the* FROM *clauses and each pair* $(\mathsf{iri_i},\mathsf{G_i})$ *results from a* FROM NAMED *clause with IRI* $\mathsf{iri_i}$ *where* $\mathsf{iri_i}$ *identifies a resource that serializes the graph* $\mathsf{G_i}$*; otherwise the dataset for* Q *is the dataset used by the queried service.*

The SPARQL abstract query *for* Q *is a tuple* $(\mathsf{algbr(Q)},\mathsf{D},\mathsf{F})$ *where* D *is the RDF dataset for* Q*, and* F *is the query form for* Q*.*

We extend the evaluation of algebra expressions as defined in Table 4. To evaluate $(\mathsf{E},\mathsf{D},\mathsf{F})$*, one first computes* $[\![\ E\ ]\!]_{\mathsf{D,G}}$*. If the query form is* SELECT*,* CONSTRUCT*, or* DESCRIBE*, the query answer is* $[\![\ E\ ]\!]_{\mathsf{D,G}}$*, which is a solution sequence for* SELECT *queries and a set of RDF triples otherwise. If the query form is* ASK *the query answer is* $\sharp([\![E]\!]_{\mathsf{D,G}}) > 0$*.*

Note that in case of DESCRIBE queries, the concrete result is not normatively defined and depends on the implementation.

Example 17 In order to see some examples of abstract queries, we go through several of the examples again, in particular those with solution modifiers or query forms other than SELECT.

Example 1:
Abstract Query: (Project(ToList(algbr(P)), {?name, ?mbox}), D, SELECT)
with P the query pattern from Example 1 and D the dataset of the SPARQL query processor.

Example 2
Abstract Query:

(Project(ToList(algbr(P)), {?name, ?mbox}), {∅, (iri, G_{iri})}, SELECT)
with P the query pattern from Example 2, iri the IRI in the FROM NAMED clause, and G_{iri} the graph serialized by iri.

Example 11
Abstract Query: (Construct(ToList(algbr(P)), templ), D, CONSTRUCT)
with P the query pattern from Example :ex : construct,templ the template from the query, and D the dataset of the SPARQL query processor.

Example 12
Abstract Query: (ToList(algbr(P)), D, ASK)
with P the query pattern from Example :ex : ask and D the dataset of the SPARQL query processor.

Example 13
Abstract Query (a): (Describe(ToList(Z), {foaf:Person}), D, DESCRIBE)
Abstract Query (b): (Describe(ToList(algbr(P)), {?x}), D, DESCRIBE)
with P the query pattern from the second query in Example :ex : describe and D the dataset of the SPARQL query processor.

Example 14
Abstract Query:
(Project(OrderBy(ToList(algbr(P)), (ASC(?name))), {?name}), D, SELECT)
with P the query pattern from Example :ex : order and D the dataset of the SPARQL query processor.

2.5 SPARQL 1.1 Features

The W3C Data Access Working Group is currently in the process of specifying the next version of SPARQL, which is named SPARQL 1.1. The new version adds several features to the query language [13], which we briefly introduce in this section. We do not provide an algebra translation for these features and only give examples of how these features can be used. We further give a brief overview of the new parts in SPARQL 1.1 apart from the query language and the entailment regimes.

Aggregates Aggregates allow for counting the number of answers, computing average, minimal, or maximal values from solutions by applying expressions ver groups of solutions.

Example 18 One intuitive way of aggregates is counting, which we illustrate with the data from Table 1 and the query:

```
SELECT (COUNT(?x) AS ?num)
WHERE  { ?x foaf:name ?name }
```

We obtain one solution with binding 3 for ?num.

Table 5. Triples used in the examples for the new features of SPARQL 1.1

```
:auth1  :writes :book1 .
:auth1  :writes :book2 .
:auth2  :writes :book3 .
:auth3  :writes :book4 .
:book1 :costs  9 .
:book2 :costs  5 .
:book3 :costs  11 .
:book4 :costs  2 .
```

Example 19 In order to select variables to which no aggregate is applied, one has to group the solutions accordingly. We illustrate this with the data from Table 5 and the query:

```
SELECT      ?auth (AVG(?price)AS?avgPrice)
WHERE       { ?auth :writes ?book . ?book:costs ?price }
GROUP BY ?auth
```

Without grouping by author, we were not able to have ?auth in the SELECT clause. We obtain:

?auth	?avgPrice
:auth1	7
:auth2	11
:auth2	2

We can further extend the query with a HAVING clause to filter some of the aggregated values. For example, adding

```
HAVING (AVG(?price) > 5)
```

results in the last solution being filtered out.

Subqueries Subqueries provide a way to embed SPARQL queries within other queries, normally to achieve results which cannot otherwise be achieved

Example 20 We use again the triples from Table 5. We use a subquery to answer the query "Which author has a book that costs more than the most expensive book of :auth1?". For such query, two queries have to be executed: the first query finds the most expensive book of :auth1 and the second finds those authors who have a book more expensive than that. Using subqueries we can directly embed the first query into the second:

```
SELECT ?auth
WHERE  { ?auth :writes ?book . ?book :costs ?price
           {
              SELECT (MAX(?price) AS ?max)
              WHERE { :auth1 :writes ?book . ?book :costs ?price }
           }
           FILTER (?price > ?max)
         }
```

Evaluating the inner query yields 9 as binding for ?max. Note that only ?max is projected and, therefore, available to the outer query. The variable ?price from the inner query is not visible for the outer query (it is *out of scope*) and the variable ?price from the outer query unrelated to it. The filter applies, as usual, to all elements in the group, which makes sure that the two triple patterns and the subquery are evaluated and the results are joined before the filter is applied.

Negation comes in two styles, one is based on filtering out results that do not match a given graph pattern using filers with the NOT EXISTS keyword, and the other way is to directly remove solutions related to another pattern with MINUS.

Filtering of query solutions is done within a FILTER expression using NOT EXIST and EXISTS.

Example 21 We illustrate the use of filtering combined with NOT EXISTS (EXISTS) using the data from Table 1 and the query:

```
SELECT ?name
WHERE { ?x foaf:name ?name
          FILTER NOT EXISTS { ?x foaf:mbox ?mbox }
      }
```

Since only for ?x bound to "Pascal Hitzler" the NOT EXISTS filter evaluates to true, we get just one solution:

?name
"Pascal Hitzler"

We can similarly test for the existence of a match for the pattern by using FILTER EXISTS instead of FILTER NOT EXISTS, which would yield one solution with ?name once bound to "Birte Glimm" and once to "Sebastian Rudolph".

A different form of negation is supported via the MINUS keyword, which takes the form pattern MINUS pattern.

Example 22 We illustrate the use MINUS with the following query over the data from Table 1:

```
SELECT ?name ?mbox
WHERE { ?x foaf:name ?name . ?x foaf:mbox ?mbox
          MINUS { ?x foaf:name "Birte Glimm" }
      }
```

In this case, the left-hand side consists of the two triple patterns, which yield the two solutions:

?x	?name	?mbox
_:a	"Birte Glimm"	"b.glimm@googlemail.com"
_:b	"Sebastian Rudolph"	<mailto:rudolph@kit.edu>

The right-hand side of the MINUS operator yields one solution in which ?name and ?mbox are unbound (since they do not occur in the pattern and are, therefore, not matched):

?x	?name	?mbox
_:a		

In order to compute the query result, we keep only solutions for the left-hand side pattern if they are not "compatible" with the solutions for the right-hand side. Two mappings are compatible if whenever they both map a variable, then they map it to the same value. We will refer to the two solutions for the left-hand side pattern as l_1 and l_2, respectively, and to the solution for the right-hand side pattern as r_1. In our case, we have that l_1 is compatible with r_1 since the mappings for ?x is the same and since ?name and ?mbox are unbound in r_1, which does not contradict the mapping for ?name and ?mbox in l_1. Since l_1 and r_1 are compatible, l_1 is removed from the solutions. For l_2 and r_1, it is, however, clear that the mappings are not compatible since l_2 maps ?x to _:b whereas r_1 maps ?x to _:a. This means that l_2 remains in the solutions for the whole pattern, which gives the following overall result:

?name	?mbox
"Sebastian Rudolph"	<mailto:rudolph@kit.edu>

SELECT Expressions can be used in the SELECT clause to combine variable bindings already in the query solution, or defined earlier in the SELECT clause to produce a binding in the query solution, e.g., SELECT ?net ((?net * 1.2) AS ?gross) will return bindings for ?net and 120% of that value as binding for ?gross.

Property Paths allow for specifying a possible route through a graph between two graph nodes through property path expressions that apply to the predicate of a triple. A trivial case is a property path of length exactly 1, which is a triple pattern. Property paths allow for more concise expression of some SPARQL basic graph patterns and also add the ability to match arbitrary length paths. For example, the BGP

?x foaf:name "Birte Glimm" . ?x foaf:knows+/foaf:name ?name

The query starts from an element which is associated with the name Birte Glimm and then follows a path of length one or more (+) along the property foaf:knows followed by (/) a path of length one along the foaf:name property. Thus, the query finds the names of all people that Birte knows directly or indirectly. The / operator can always be eliminated, e.g., by rewriting the second triple into ?x foaf:knows + ?y . ?y foaf:name ?name. The arbitrary length paths, however, cannot be eliminated in this way and a SPARQL query processor has to implement them natively in order to fully support property paths.

Assignments can be used in addition to SELECT expressions in order to add bindings. Whereas SELECT expressions are limited to the SELECT clause, on can use the BIND keyword followed by an expression to add bindings already in the WHERE clause. The BINDINGS keyword can further be used to provide a solution sequence that is to be joined with the query results. It can used by an application to provide specific requirements on query results and also by SPARQL query engine implementations that provide federated query through the SERVICE keyword to send a more constrained query to a remote query service.

CONSTRUCT Short Forms allow for abbreviating CONSTRUCT queries provided the template and the pattern are the same and the pattern is just a basic graph pattern (i.e., no FILTERs and no complex graph patterns are allowed in the short form). The keyword WHERE is required in the short form.

Furthermore, SPARQL 1.1 provides an expanded set of functions and operators.

SPARQL 1.1 Update [28] provides a way of modifying a graph store by inserting or deleting triples from graphs. It provides the following facilities:

– Insert new triples into an RDF graph.
– Delete triples from an RDF graph.
– Perform a group of update operations as a single action.
– Create a new RDF graph in a graph store.
– Delete an RDF graph from a graph store.

The behavior of update queries in a system that uses entailment regimes is left open in SPARQL 1.1. A straightforward way of implementing updates under entailment regimes would be to interpret such queries under simple entailment semantics.

The SPARQL protocol has also been extended to allow for an exchange of update requests between a client and a SPARQL endpoint [22].

Service Descriptions [30] have been added SPARQL 1.1 as a method for discovering and vocabulary for describing SPARQL services made available via the SPARQL Protocol for RDF[11]. Such a description is intended to provide a mechanism by which a client or end user can discover information about the SPARQL implementation/service such as supported extension functions and details about the available dataset or the used entailment regime.

Federation Extensions are currently under development as part of SPARQL 1.1 to express queries across distributed data sources. At the time of writing a first public working draft is available at `http://www.w3.org/TR/sparql11-federated-query/`.

JSON Result Format is so far a working group note available at `http://www.w3.org/TR/rdf-sparql-json-res/`. The working group intends to bring this to recommendation status, but at the time of writing no public working draft is available.

3 SPARQL Entailment Regimes

In the previous section, we have defined the syntax of SPARQL queries and how such queries are evaluated with subgraph matching as means of evaluating basic graph patterns. This form of basic graph pattern evaluation is also called simple entailment since it can equally be defined in terms of the simple entailment relation between RDF graphs. In order to use more elaborate entailment relations, which also allow for retrieving solutions that implicitly follow from the queried graph, we now look at so-called *entailment regimes*. An entailment regime specifies how an entailment relation such as RDF Schema entailment can be used to redefine the evaluation of basic graph patterns from a SPARQL query making use of SPARQL's extension point for basic graph pattern matching. In order to satisfy the conditions that SPARQL places on extensions to basic graph pattern matching, an entailment regimes specifies conditions that limit the number of entailments that contribute solutions for a basic graph pattern. For example, only a finite number of the infinitely many axiomatic triples can contribute solutions under RDF Schema (RDFS) entailment. In this section, we introduce the RDFS entailment regime and explain the design rationale behind the regime. In Section 4, we then show how the OWL 2 Direct Semantics entailment relation can be used.

Each entailment regime is characterized by a set of properties:

Name:	A name for the entailment regime, usually the same as the entailment relation used to define the evaluation of a basic graph pattern.
IRI:	The IRI for the regime. This IRI can be used in the service description for a SPARQL endpoint, which is an RDF graph that describes the functionality and the features that it provides.
Legal Graphs:	Describes which graphs are legal for the regime.
Legal Queries:	Describes which queries are legal for the regime.
Illegal Handling:	Describes what happens in case of an illegal graph or query.
Entailment:	Specifies which entailment relation is used in the evaluation of basic graph patterns.
Inconsistency:	Defines what happens if the queried graph is inconsistent under the used semantics.
Query Answers:	Defines how a basic graph pattern is evaluated, i.e., what the solutions are for a given graph and basic graph pattern of a query.

Before we start describing a concrete entailment regime, we first analyze what conditions an entailment regime has to satisfy. These conditions also motivate the choice of the above properties that are defined for each entailment regime.

3.1 Conditions on Extensions of Basic Graph Pattern Matching

In order to extend SPARQL for an entailment relation E such as RDFS or OWL Direct Semantics entailment, it suffices to modify the evaluation of BGPs accordingly, while the remaining algebra operations can still be evaluated as in Definition 7. When considering E-entailment, we thus define solution multisets $[\![\mathsf{BGP}]\!]^{\mathsf{E}}_{\mathsf{D,G}}$.

The SPARQL Query 1.0 specification [26] already envisages the extension of the BGP matching mechanism, and provides a set of conditions for such extensions that we recall in Table 6. These conditions can be hard to interpret since their terminology is not

Table 6. Conditions for extending BGP matching to E-entailment (quoted from [26])

1. The scoping graph SG, corresponding to any consistent active graph AG, is uniquely specified and is E-equivalent to AG.
2. For any basic graph pattern BGP and pattern solution mapping P, P(BGP) is well-formed for E.
3. For any scoping graph SG and answer set $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ for a basic graph pattern BGP, and where $\mathsf{BGP}_1, \ldots, \mathsf{BGP}_n$ is a set of basic graph patterns all equivalent to BGP, none of which share any blank nodes with any other or with SG
$$\mathsf{SG} \models_{\mathsf{E}} (\mathsf{SG} \cup \mathsf{P}_1(\mathsf{BGP}_1) \cup \ldots \cup \mathsf{P}_n(\mathsf{BGP}_n)).$$
4. Each SPARQL extension must provide conditions on answer sets which guarantee that every BGP and AG has a finite set of answers which is unique up to RDF graph equivalence.

Table 7. Clarified conditions for extending BGP matching to E-entailment

An entailment regime E must provide conditions on basic graph pattern evaluation such that for any basic graph pattern BGP, any RDF graph G, and any evaluation $[\![\cdot]\!]^{\mathsf{E}}_{\mathsf{G}}$ that satisfies the conditions, the multiset of graphs $\{(\mu(\mathsf{BGP}), n) \mid (\mu, n) \in [\![\mathsf{BGP}]\!]^{\mathsf{E}}_{\mathsf{G}}\}$ is uniquely determined up to RDF graph equivalence. An entailment regime must further satisfy the following conditions:

1. For any consistent active graph AG, the entailment regime E uniquely specifies a *scoping graph* SG that is E-equivalent to AG.
2. A set of *well-formed* graphs for E is specified such that, for any basic graph pattern BGP, scoping graph SG, and solution mapping μ in the underlying set of $[\![\mathsf{BGP}]\!]^{\mathsf{E}}_{\mathsf{SG}}$, the graph $\mu(\mathsf{BGP})$ is well-formed for E.
3. For any basic graph pattern BGP, and scoping graph SG, if $\{\mu_1, \ldots, \mu_n\} = [\![\mathsf{BGP}]\!]^{\mathsf{E}}_{\mathsf{SG}}$ and $\mathsf{BGP}_1, \ldots, \mathsf{BGP}_n$ are basic graph patterns all equivalent to BGP but not sharing any blank nodes with each other or with SG, then
$$\mathsf{SG} \models_{\mathsf{E}} \mathsf{SG} \cup \bigcup_{1 \leq i \leq n} \mu_n(\mathsf{BGP}_n).$$
4. Entailment regimes *should* provide conditions to prevent trivial infinite solution multisets.

aligned well with the remaining specification. In the following, we discuss our reading of these conditions, leading to a revised clarified version presented in Table 7.[2]

Condition (1) forces an entailment regime to specify a scoping graph based on which query answers are computed instead of using the active graph directly. Since an entailment regime's definition of BGP matching is free to refer to such derived graph structures anyway, the additional use of a scoping graph does not increase the freedom of potential extensions. We assume, therefore, that the scoping graph is the active graph in the remainder. If the active graph is E-inconsistent, entailment regimes specify the intended behavior directly, e.g., by requiring that an error is reported.

Condition (2) refers to a "pattern solution mapping" though what is probably meant is a *pattern instance mapping* P, defined in [26] as the combination of an RDF instance mapping σ and a solution mapping μ where $\mathsf{P}(x) = \mu(\sigma(x))$. We assume, however, that

[2] The SPARQL 1.1 Query working draft has been updated to contain the revised conditions.

(2) is actually meant to refer to all solution mappings in $[\![\mathsf{BGP}]\!]^{\mathsf{E}}_{\mathsf{D,G}}$. Indeed, even for simple entailment where well-formedness only requires $\mathsf{P}(\mathsf{BGP})$ to be an RDF graph, condition (2) would be violated when using *all* pattern instance mappings. To see this, consider a basic graph pattern

$$\{\ \_{:}a\ {:}b\ {:}c\ \}.$$

Clearly, there is a pattern instance mapping P with $\mathsf{P}(\_{:}a) = \texttt{"1"\^\^}xsd{:}int$, but $\mathsf{P}(\mathsf{BGP}) = \{\texttt{"1"\^\^}xsd{:}int\ {:}b\ {:}c\}$ is not an RDF graph. Similar problems occur when using all solution mappings. Hence we assume (2) to refer to elements of the computed solution multiset $[\![\mathsf{BGP}]\!]^{\mathsf{E}}_{\mathsf{D,G}}$. The notion of *well-formedness* in turn needs to be specified explicitly for entailment regimes.

Condition (3) uses the term "answer set" to refer to the results computed for a BGP. To match the rest of [26], this has to be interpreted as the solution multiset $[\![\mathsf{BGP}]\!]^{\mathsf{E}}_{\mathsf{D,G}}$. This also means mappings P_i are solution mappings (not pattern instance mappings as their name suggests). The purpose of (3), as noted in [26], is to ensure that if blank node names are returned as bindings for a variable, then the same blank node name occurs in different solutions only if it corresponds to the same blank node in the graph.

Example 23 To illustrate the problem, consider the following graphs:

G : :a :b _:c.	G_1 : :a :b _:b$_1$.	G_2 : :a :b _:b$_2$.	G_3 : :a :b _:b$_1$.
_:d :e :f.	_:b$_2$:e :f.	_:b$_1$:e :f.	_:b$_1$:e :f.

Clearly, G simply entails G_1 and G_2, but not G_3 where the two blank nodes are identified. Now consider a basic graph pattern BGP

$$\{\ {:}a\ {:}b\ ?x.\ ?y\ {:}e\ {:}f\ \}.$$

A solution multiset for BGP could comprise two mappings

$$\mu_1 :\ ?x \mapsto \_{:}b_1,\ ?y \mapsto \_{:}b_2 \ \text{ and}$$
$$\mu_2 :\ ?x \mapsto \_{:}b_2,\ ?y \mapsto \_{:}b_1.$$

We then have $\mu_1(\mathsf{BGP}) = \mathsf{G}_1$ and $\mu_2(\mathsf{BGP}) = \mathsf{G}_2$, and both solutions are entailed. Condition (3) requires, however, that $\mathsf{G} \cup \mu_1(\mathsf{BGP}) \cup \mu_2(\mathsf{BGP})$ is also entailed by G, and this is not the case in our example since this union contains G_3.

The reason is that our solutions have unintended co-references of blank nodes that (3) does not allow. SPARQL's basic subgraph matching semantics respects this condition by requiring solution mappings to refer to blank nodes that actually occur in the active graph, so blank nodes are treated like (Skolem) constants.[3] The revised condition in Table 7 has further been modified to not implicitly require finite solution multisets which may not be appropriate for all entailment regimes. In addition, we use RDF instance mappings for renaming blank nodes instead of requiring renamed variants of the BGP.

Finally, condition (4) requires that solution multisets are finite and uniquely determined up to RDF graph equivalence, again using the "answer set" terminology. Our

[3] Yet, SPARQL allows blank nodes to be renamed when loading documents, so there is no guarantee that blank node IDs used in input documents are preserved.

revised condition clarifies what it means for a solution multiset to be "unique up to RDF graph equivalence." We move the uniqueness requirement above all other conditions, since (2) and (3) do not make sense if the solution multiset was not defined in this sense. The rest of the condition was relaxed since entailment regimes may inherently require infinite solution multisets, e.g., in the case of the Rule Interchange Format RIF [17]. It is desirable that this only happens if there are infinite solutions that are "interesting," so the condition has been weakened to merely recommend the elimination of infinitely many "trivial" solution mappings in solution multisets. The requirement thus is expressed in an informal way, leaving the details to the entailment regime. Within this paper, we will make sure that the solution multisets are in fact finite (both regarding the size of the underlying set, and regarding the multiplicity of individual elements).

3.2 Addressing the Extension Point Conditions

Before coming to OWL, we introduce the RDFS entailment regime since RDFS is well-known and simpler than OWL while the regime still illustrates the main points in which an entailment regime differs from SPARQL's standard query evaluation. The major problem for RDFS entailment is to avoid trivially infinite solution multisets as suggested by Table 7 (4), where three principal sources of infinite query results have to be addressed:

1. An RDF graph can be inconsistent under the RDFS semantics in which case it RDFS-entails all (infinitely many) conceivable triples.
2. The RDFS semantics requires all models to satisfy an infinite number of *axiomatic triples* even when considering an empty graph.
3. Every non-empty graph entails infinitely many triples obtained by using arbitrary blank nodes in triples.

We now discuss each of these problems, and derive a concrete definition for BGP matching in the proposed entailment regime at the end of this section.

Treatment of Inconsistencies SPARQL does not require entailment regimes to yield a particular query result in cases where the active graph is inconsistent. As stated in [26], "[the] effect of a query on an inconsistent graph [. . .] must be specified by the particular SPARQL extension." One could simply require that implementations of the RDFS entailment report an error when given an inconsistent active graph. However, a closer look reveals that inconsistencies are extremely rare in RDFS, so that the requirement of checking consistency before answering queries would impose an unnecessary burden on implementations.

Indeed, graphs can only be RDFS-inconsistent due to improper use of the datatype rdf:XMLLiteral.

Example 24 A typical example for this is the following graph:

```
:a :b "<"^^rdf:XMLLiteral.        :b rdfs:range rdfs:Literal.
```

The literal in the first triple is *ill-typed* as it does not denote a value of rdf:XMLLiteral. This does not cause an inconsistency yet but forces "<"^^rdf:XMLLiteral to be interpreted as a resource that is not in the extension of rdfs:Literal, which in turn cannot be the case in any model that satisfies the second triple.

Ill-typed literals are the only possible cause of inconsistency in RDFS and as such not a frequent problem.[4] Moreover, inconsistencies of this type are inherently "local" as they are based on individual ill-typed literals that could easily be ignored if not related to a given query.

It has thus been decided in the SPARQL working group that systems only have to report an error if they actually detect an inconsistency. Until this happens, queries can be answered as if all literals were well-typed. Our exact formalization corresponds to a behavior where tools simply assume that all strings are well-typed for rdf:XMLLiteral, and hence does not put additional burden on implementers.

Treatment of Axiomatic Triples Every RDFS model is required to satisfy an infinite number of *axiomatic triples*. The reason is that the RDF vocabulary for encoding lists includes property names rdf:_i for all $i \geq 1$, with several (RDFS) axiomatic triples for each rdf:_i. For instance, we find a triple rdf:_i rdf:type rdf:Property for all $i \in \mathbb{N}$. Thus, the query ?x rdf:type rdf:Property could have infinitely many results. We consider such results trivial in the sense of Table 7 (4), and thus we want avoid them in the RDFS entailment regime.

We therefore propose that axiomatic triples with a subject of the form rdf:_i are only taken into account if the subject's IRI explicitly occurs in the active graph. This ensures that only finitely many axiomatic triples are considered, since there is only a finite number of axiomatic triples whose subjects do not have the form rdf:_i. To conveniently formalize this, Definition 10 below still refers to the standard RDFS entailment, but restricts the range of solution mappings to a *finite* vocabulary, which consists of terms from the queried graph and from terms of the RDFS vocabulary apart from those of the form rdf:_i.

Treatment of Blank Nodes Even if condition (3) in Table 7 holds, solution multisets could include infinitely many results that only differ in the identifiers for blank nodes. Simple entailment avoids this problem by restricting results to blank nodes that occur in the active graph. For entailment regimes, however, one must take entailed triples into account. This already leads to triples with different blank nodes, as illustrated in the graphs G_1 and G_2 in Example 23.

Restricting the range of solution mappings to blank nodes in the active graph would ensure finiteness but is not a satisfactory solution.

[4] Implementations may support additional datatypes that can lead to similar problems. Such extensions go beyond the RDFS semantics we consider here, yet inconsistencies remain rare even in these cases.

Example 25 To see why restricting the range of solution mappings to blank nodes in the active graph is not a satisfactory, consider the graph

G : :a :b :c.   :d :e _:f.

The query pattern BGP = { :a :b ?x } yields only one solution mapping μ : ?x $\mapsto$:c under simple entailment. Yet, the mapping μ' : ?x $\mapsto$ _:f uses only blank nodes from G, and satisfies G $\models$ μ'(BGP) even under simple semantics.

This shows that the latter two conditions are not sufficiently specific for handling blank nodes in entailment regimes. A more adequate approach is the use of *Skolemization*:

Definition 9. *Let the prefix* skol *refer to a namespace IRI that does not occur as the prefix of any IRI in the active graph or query. The* Skolemization sk(_:b) *of a blank node* _:b *is defined as* sk(_:b) := skol:b. *We extend* sk($\cdot$) *to graphs and filters just like other (partial) functions on RDF terms.*

Intuitively, Skolemization changes blank nodes into resource identifiers that are not affected by entailment. Clearly, we do not want Skolemized blank nodes to occur in query results, but it is useful to restrict to solution mappings μ for which sk(G) $\models$ sk(μ(BGP)). In Example 25 above, this condition is indeed satisfied by μ but not by μ'.

In order to illustrate the effect, we use an RDF graph that does not make use of any special RDFS terms, i.e., simple entailment would result in the same solutions. Let G, sk(G), and BGP be as follows:

```
G : :a :b :c.        sk(G) : :a :b :c.            BGP : ?x  :b  _:d
    _:a :b _:c.              skol:a :b skol:c.
```

Here the Skolem function sk maps _:a to skol:a and _:c to skol:c for skol defined as some imaginary prefix not used anywhere in G or BGP. We can now return only those solutions μ for which applying the Skolem function to blank nodes in the range of μ and some RDF instance mapping σ yields ground triples that are entailed by sk(G). For example, all the mappings below yield entailed triples, but only the first two satisfy the stated requirement because applying sk to _:a and _:c yields a ground triple that is entailed by sk(G):

$$\begin{array}{ll} \mu_1 : \text{?x} \mapsto \text{:a} & \sigma_1 : \text{\_:d} \mapsto \text{:c} \\ \mu_2 : \text{?x} \mapsto \text{\_:a} & \sigma_2 : \text{\_:d} \mapsto \text{\_:c} \\ \mu_3 : \text{?x} \mapsto \text{\_:b}_1 & \sigma_3 : \text{\_:d} \mapsto \text{\_:b}_2 \end{array}$$

3.3 The RDFS Entailment Regime

The set of *well-formed* graphs for the RDFS entailment regime is simply the set of all RDF graphs. BGP matching for RDFS is defined as follows.

Definition 10. *Let* G *be an RDF graph,* BGP *a basic graph pattern,* V(BGP) *the set of variables in* BGP, B(BGP) *the set of blank nodes in* BGP, sk *a Skolemization function* *as in Definition 9 such that* $\mathsf{ran}(\mathsf{sk}) \cap (\mathsf{Voc}(\mathsf{G}) \cup \mathsf{Voc}(\mathsf{BGP})) = \emptyset$. *Let* Voc(RDFS) *be the RDFS vocabulary and* $\mathsf{Voc}^{-}(\mathsf{RDFS}) = \mathsf{Voc}(\mathsf{RDFS}) \setminus \{\mathsf{rdf{:}\_i} \mid i \in \mathbb{N}\}$.

We write $\models_{\mathsf{RDFS}}$ *for the RDFS entailment relation and define the* evaluation of BGP over G under RDFS entailment, $[\![\mathsf{BGP}]\!]^{\mathsf{RDFS}}_{\mathsf{D,G}}$, *as the solution multiset*

$\{(\mu, n) \mid \mathsf{dom}(\mu) = \mathsf{V}(\mathsf{BGP})$, *and n is the maximal number of distinct RDF instance mappings such that, for each* $1 \leq i \leq n$,

(i) $\mathsf{dom}(\mathsf{sigma_i}) = \mathsf{B}(\mathsf{BGP})$,
(ii) $\mu(\sigma_i(\mathsf{BGP}))$ *are well-formed RDF triples*,
(iii) $\mathsf{sk}(\mu(\sigma_i(\mathsf{BGP})))$ *are ground RDF triples*,
(iv) $\mathsf{sk}(\mathsf{G}) \models_{\mathsf{RDFS}} \mathsf{sk}(\mu(\sigma_i(\mathsf{BGP})))$, *and*
(v) $\mathsf{ran}(\mu) \subseteq \mathsf{Voc}(\mathsf{G}) \cup \mathsf{Voc}^{-}(\mathsf{RDFS})\}$.

Other types of graph patterns are evaluated as in Definition 7. If the active graph is RDFS-inconsistent, implementations may compute solution multisets based on the assumption that all literals of type rdf:XMLLiteral are well-typed, so that no inconsistency occurs. When the inconsistency is detected, implementations should report an error. We summarize the RDFS entailment regime in Table 8.

Condition (*i*) ensures that only RDF instance mappings that map all and only the blank nodes of BGP can increase the multiplicity of a solution mapping. Condition (*ii*) ensures that the instantiated triples are well-formed, e.g., variables tat occur in the subject position cannot be mapped to a literal by a solution mapping. Similarly, variables in the predicate position cannot be mapped to blank nodes. Condition (*iii*) then ensures that all blank nodes are indeed Skolemized by sk, resulting in ground RDF triples. Condition (*iv*) and (*v*) ensure that blank nodes and the axiomatic triples are handled as described in the previous section, therefore, avoiding infinitely many answers.

The definition might look quite complicated, but has the advantage that we can simply swap in another entailment relation and vocabulary to get another entailment regime. For example, when we use the simple entailment relation in place of the RDFS entailment relation and the empty set instead of Voc(RDFS) (as there are no special terms for simple interpretations), then we get exactly the behavior of subgraph matching (aka simple entailment) described in Definition 6. Furthermore, we can also swap the RDFS entailment relation for RDF or the OWL RDF-Based Semantics entailment relation and get a valid entailment regime. The OWL Direct Semantics needs some minor tweaks as the Direct Semantics is not defined in terms of triples, but based on Description Logics.

Example 26 In order to see why the range of a solution mapping can also use terms from Voc(RDFS), we consider the data from Table 1 and the query:

```
SELECT ?name
WHERE { ?x foaf:name ?name . ?x rdf:type foaf:Person }
```

Under RDFS entailment, the queried graph entails

```
_:a  foaf:name  "Birte Glimm"
_:a  rdf:type   foaf:Person
```

Thus, μ_1 : ?name $\mapsto$ "Birte Glimm" is a solution. Note, however, that rdf:type is not part of the vocabulary of the graph, and the solution is only part of the result since we include the RDFS vocabulary. Overall, we get the following three solutions:

?**name**
"Birte Glimm"
"Sebastian Rudolph"
"Pascal Hitzler"

Furthermore, in order to implement the regime, we can simply materialize all RDFS inferences and use subgraph matching on the extended graph. We illustrate this with the next example.

Example 27 In order to get an idea of how we can implement the RDFS entailment regime via materialization, we consider again the data from Table 1 and the query from the previous example.

In order to materialize all RDFS inferences, we add triples that are RDFS entailed and obtain a graph G′, which contains (among other triples):

_:a	rdf:type	foaf:Person
_:b	rdf:type	foaf:Person
_:c	rdf:type	foaf:Person

due to the triple foaf:name rdfs:domain foaf:Person combined with the three triples with the predicate foaf:name. Furthermore, we would add

_:a	foaf:nick	foaf:b.glimm
_:c	foaf:nick	foaf:phi

due to the fact that foaf:icqChatID is a subproperty of foaf:nick. Furthermore, the full materialization would also contain triples such as t rdf:type rdfs:Resource, for each term t in subject or object position plus other triples (cf. [14], [15]).

For evaluation the query, we do not have to make the Skolemization explicit, instead, we can just consider the blank nodes in G′ as constants. However, if a blank node occurs in the query that occurs also in the graph, we have to keep in mind that the blank node from the query cannot only map to that very blank node in the graph, but it still acts like a variable. Thus, if _:x in our query were _:a, it could still match to _:b in G′. Hence, we get the same three solution by performing subgraph matching on G′ as in the previous example.

Since computing the required partial RDFS closure (partial, since we do not require all axiomatic triples) can be done in polynomial time [15] and BGP evaluation then amounts to subgraph matching over the partial closure, it follows that the complexity of the evaluation problem under the RDFS regime is the same as for standard SPARQL. For set semantics instead of multiset semantics this is known to be PSPACE-complete [24].

Table 8. The RDFS entailment regime

Name	RDFS
IRI	http://www.w3.org/ns/entailment/RDFS
Legal Graphs	Any legal RDF graph
Legal Queries	Any legal SPARQL query
Illegal Handling	In case the query is illegal (syntax errors), the system must raise a MalformedQuery fault. In case the queried graph is illegal (syntax errors), the system must raise a QueryRequestRefused fault.
Entailment	RDFS Entailment
Inconsistency	The scoping graph is graph-equivalent to the active graph even if the active graph is RDFS-inconsistent. If the active graph is RDFS-inconsistent, an implementation may raise a QueryRequestRefused fault or issue a warning and it should generate such a fault or warning if, in the course of processing, it determines that the data or query is not compatible with the request. In the presence of an inconsistency the conditions on solutions still guarantee that answers are finite.
Query Answers	Basic Graph Patterns are evaluated as in Definition 10

4 The OWL Entailment Regimes

In contrast to the RDFS semantics, a graph does no longer admit a unique canonical model that can be used to compute answers under the RDF-Based Semantics (RBS) and Direct Semantics (DS) of OWL, i.e., we can no longer imagine queries to act on a unique "completed" version of the active graph. This affects reasoning algorithms, but has only little effect on our definitions. The main new challenges for OWL are its expressive datatype constructs that may lead to infinite answers, and the fact that the OWL DS is defined in terms of OWL objects to which a given RDF graph and query must first be translated. The problems discussed for RDF(S) also require slightly different solutions for OWL:

1. Inconsistent input ontologies are required to be rejected with an error.
2. The axiomatic triples of RDFS are used only by the RBS and can again be handled by suitably restricting solutions to terms from a finite vocabulary.
3. The problem of blank nodes occurs for both semantics and can again be addressed by Skolemization, but for DS the blank nodes that are used to encode OWL objects must not be Skolemized.

The main difference to RDFS is the stricter first item which no longer permits deferred inconsistency detection. Inconsistencies in RDFS were easy to ignore since they always related to single literals. Neither OWL semantics suggests such simple reasoning under inconsistencies. Although proposals exists for addressing this, they disagree on the inferred entailments and tend to require complex computations. On the other hand, typical OWL reasoning algorithms are model building procedures which detect inconsistencies as part of their normal operation. Hence, reporting errors in this case can usually be done without additional effort.

4.1 Mapping from RDF Graphs to OWL Structural Objects

For the OWL 2 Direct Semantics entailment regime, semantic conditions are defined with respect to ontology structures (i.e., instances of the Ontology class as defined in the OWL 2 structural specification [21]). Given an RDF graph G, the ontology structure for G, denoted O_G, is obtained by mapping the queried RDF graph into an OWL 2 ontology [23]. This mapping is only defined for OWL 2 DL ontologies, i.e., ontologies that satisfy certain syntactic conditions.

In this section, we use both Turtle and OWL's functional-style syntax (FSS) that is used in the OWL 2 structural specification [21]. We further provide a Description Logic (DL) syntax version for those with a background in DLs.

For many triples that use as predicate a special term from the RDFS vocabulary, the mapping to OWL structural objects is straightforward.

Example 28 For example a subclass statement in RDFS has a straightforward representation in OWL's FSS:

Turtle:	foaf:Person rdfs:subClassOf foaf:Agent .		
FSS:	SubClassOf(foaf:Person foaf:Agent)		
DL:	Person	$\sqsubseteq$	Agent

Note that DLs have no notion of IRIs, namespaces, or prefix declaration and we just write the short name without any prefix in the DL syntax. It is also characteristic that several terms of the specialized RDFS and OWL vocabulary in the Turtle syntax are translated to constructors in the FSS, e.g., rdfs:subClassOf is mapped into a SubClassOf constructor.

Similarly, the translation of domains and ranges is relatively straightforward.

Example 29 For example, the following domain and range statements translate straightforwardly to the FSS, but the DL syntax is slightly more involved:

Turtle:	foaf:knows rdfs:range foaf:Person .		
	foaf:knows rdfs:domain foaf:Person .		
FSS:	ObjectPropertyRange(foaf:knows foaf:Person)		
	ObjectPropertyDomain(foaf:knows foaf:Person)		
DL:	$\top$	$\sqsubseteq$	$\forall$ knows Person
	$\exists$ knows.$\top$	$\sqsubseteq$	Person

First, it can be noted that in the FSS the term rdfs:range becomes ObjectPropertyRange. The counterpart to ObjectPropertyRange is DataPropertyRange range, which is used for properties that relate individuals (such as instances of the class foaf:Person) to concrete data values. For example, the property foaf:name relates an individual to a string, i.e., an element from xsd:String. Since OWL supports very expressive reasoning with datatypes, which requires different algorithms from reasoning with abstract (non-datatype) elements, every property in OWL DL must be typed. Thus, we would have that foaf:knows is of type owl:ObjectProperty whereas foaf:name is of type owl:DataProperty.

In the DL syntax, there is no direct constructor for domains and ranges. The above statements are, however, logically equivalent. The first axiom uses on the left-hand side the special symbol ⊤, which corresponds to owl:Thing and is always true. Thus, the axiom can be read as "It is always implied that all (∀) knows-successors of an element are instances of the class Person," which is exactly what a range axiom specifies. The second axiom can be read as "If an element has some (∃) knows-successor, then it is an instance of the class Person."

Elements of an OWL 2 DL Ontology Now that we have seen some examples of the mapping from RDF triples to OWL axioms, we introduce the basic elements in an OWL 2 DL ontology. An OWL 2 DL ontology consists of an *ontology header* and a set of *axioms*. The ontology header specifies the IRI of the ontology and which other ontologies are imported by it.

Example 30 The following set of RDF triples constitute a valid OWL 2 DL ontology.

Turtle:
```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
<http://example.org/ont1> rdf:type      owl:Ontology .
<http://example.org/ont1> owl:imports <http://example.org/ont2> .
```

FSS:
```
Prefix(foaf:= <http://xmlns.com/foaf/0.1/>)
Ontology(<http://example.org/ont1>
  Import(<http://example.org/ont2>)
)
```

The ontology header has no representation in Description Logic syntax and it has no direct influence on the logical consequences of the ontology other than through imports, which instruct an OWL parser to additionally include the triples that are obtained from parsing the imported ontology.

The axioms in an ontology are used to describe a domain of interest, e.g., in the previous section we described people, their names, email addresses and chat IDs making use of terms from the FOAF (Friend of a Friend) ontology. Within the axioms, we distinguish between *logical* and *non-logical* axioms. As the ontology header, non-logical axioms carry no semantics, i.e., they do not influence the consequences of an ontology, and include:

– Annotations,
– Entity Declarations

With ontology annotations, one can describe properties of the ontology, e.g., who created it, which version of the ontology this is and other things. Similarly, one can annotate other axioms, e.g., with a comment or with provenance information, and one can even annotate annotations themselves. Entity declarations specify the types of terms. For example, we have learned above that foaf:knows is an object property whereas foaf:name is a data property. In addition to object and data properties, OWL also provides recognizes annotation properties, e.g., rdfs:label or rdfs:comment are built-in annotation properties, but one can define additional custom ones too. Similarly one can declare classes and custom datatypes (ones that are not defined in the OWL 2 datatype map) and named individuals. Such declarations are required to allow for an unambiguous parsing process.

Example 31 We can extend the ontology from Example 30 with the following annotations and declaration. Since the axioms are non-logical, the extended ontology still only entails tautological statements under the Direct Semantics.

```
Turtle:  <http://example.org/ont1> owl:priorVersion <http://example.org/ont0> .
         foaf:knows                rdf:type         owl:ObjectProperty .
         <http://example.org/ont1> rdfs:label       "An example" .

   FSS:   Annotation(owl:priorVersion <http://example.org/ont0>)
          Annotation(rdfs:label "An example")
          Declaration(ObjectProperty(foaf:knows))
```

The first annotation gives the IRI of a previous version for the current ontology and the second annotation just provides a label for the ontology. The declaration axiom specifies foaf:knows as an object property.

In the remainder we frequently omit type declarations. Unless otherwise specified, examples assume that properties are object properties and that terms refer to classes rather than data ranges.

Complex Classes and Axioms So far we always had a straightforward correspondence between one triple and one OWL axiom. A FSS axiom can, however, correspond to several RDF triples, and the RDF triples might contain auxiliary blank nodes that are not part of the corresponding OWL objects and are not visible in the corresponding FSS axiom. This is usually the case if we want to represent complex OWL classes in RDF triples. In most cases, we can "hide" the blank nodes and obtain a slightly more readable Turtle format by making use of Turtles's abbreviations: [...] implicitly introduces a blank node, ";" can be used if the following triple has the same subject, which is them omitted, "," acts as ";" but for the case where triples share subject and object, the (...) constructor abbreviates lists of terms, and a abbreviates rdf:type.

Example 32 The first class assertion uses just a class name, which requires a single RDF triple, but the second assertion uses a complex class, which requires several RDF triples with auxiliary blank nodes.

```
          Turtle:  :Peter rdf:type             :Person .
                   :Peter rdf:type             _:x .
                   _:x    rdf:type             owl:Restriction .
                   _:x    owl:onProperty       :hasFather .
                   _:x    owl:someValuesFrom   :Person .

   Turtle (abbr.):  :Peter a :Person .
                    :Peter a [ a  owl:Restriction  ;
                               owl:onProperty  :hasFather ;
                               owl:someValuesFrom   :Person ] .

             FSS:  ClassAssertion(:Person :Peter)
                   ClassAssertion(ObjectSomeValuesFrom(:hasFather :Person) :Peter)

              DL:  Person(Peter)
                   (∃ hasFather.Person)(Peter)
```

The first axiom just states that the individual :Peter is an instance of the class :Person. The second axiom states that :Peter belongs to the class of things that have a :hasFather-successor which is an instance of the class :Person.

Example 33 Disjunctions and conjunctions in the FSS similarly require several triples in RDF:

```
Turtle:         :Birte rdf:type      _:x .
                _:x    rdf:type      owl:Class .
                _:x    owl:unionOf   _:l1 .
                _:l1   rdf:first     :Vegetarian .
                _:l1   rdf:next      _:l2 .
                _:l2   rdf:first     :Vegan .
                _:l2   rdf:rest      rdf:nil .
Turtle (abbr.): :Birte a [ a owl:Class ; owl:unionOf ( :Vegetarian :Vegan ) ] .
FSS:            ClassAssertion(ObjectUnionOf(:Vegetarian :Vegan) :Birte)
```

DL: $\text{Birte} \sqsubseteq \text{Vegetarian} \sqcup \text{Vegan}$

The typing as owl:Class is required since owl:unionOf can equally be used to build the union of two datatypes or data ranges (i.e., complex datatypes that are already obtained by combining datatypes). Axiom states that the individual :Birte is a vegan or a vegetarian, i.e., an instance of the class ObjectUnionOf(:Vegan :Vegetarian).

Blank Nodes and Anonymous Individuals Although in the above examples it was always the case that the blank nodes disappeared in the FSS, this is not always the case. The FSS may still contain blank nodes, but these correspond to OWL individuals that have no explicit names and are called *anonymous individuals*.

Example 34 The following axiom uses anonymous individuals:

```
Turtle: :Peter :hasBrother _:y .
FSS:    ObjectPropertyAssertion(:hasBrother :Peter _:y)
```

The meaning of the axiom is exactly the same as the meaning of the second axiom from Example 32, i.e., we say that :Peter is related to *some* element with the property :hasBrother. Note that in DL notation there is no counterpart to anonymous individuals and one always has to use existential quantifiers (∃) as in the first version of this axiom. For RDF graphs that can be mapped into OWL 2 DL ontologies, it is, however, guaranteed that an according DL version always exists.

While parsing an input document (containing RDF triples) into an OWL ontology, it can be necessary to rename blank nodes/anonymous individuals and there is no guarantee that the blank node identifier _:y from the above triple is used as an identifier for Peter's brother in the ontology structure. Thus, the latter axiom from Example 34 could also be parsed as the OWL axiom

ObjectPropertyAssertion(:hasBrother :Peter _:somethingelse)

Table 9. RDF data for Example 35

```
(1)   <http://example.org/myOntology> a owl:Ontology
(2)   :eats a owl:ObjectProperty
(3)   :contains a owl:ObjectProperty
(4)   :Vegetarian a owl:Class
(5)   :Vegan a owl:Class
(6)   :MilkProduct a owl:Class
(7)   :Birte a [ a owl:Class ; owl:unionOf ( :Vegetarian :Vegan ) ] .
(8)   :Birte :eats :Yoghurt .
(9)   :Yoghurt :contains :Milk .
(10)  :Milk a :MilkProduct .
(11)  [ a owl:Restriction ; owl:onProperty :contains ; owl:someValuesFrom :MilkProduct ]
          rdfs:subClassOf :MilkProduct .
(12)  :Vegan rdfs:subClassOf
          [ a owl:Restriction ; owl:onProperty :eats ; owl:allValuesFrom
              [a owl:Class ; owl:complementOf :MilkProduct ]
          ]
```

4.2 Introduction to the OWL Direct Semantics for SPARQL

Having introduced the basic ideas of how we get from an RDF graph to an ontology that can be interpreted under OWL's Direct Semantics, we now turn our attention to the issue of deciding what is a consequence of an OWL ontology and how we can query for such consequences with SPARQL.

OWL Entailment OWL reasoners are tools that decide OWL entailment. In order to decide whether an RDF graph G entails an RF graph G' under OWL 2 Direct Semantic entailment, we can proceed as follows:

1. We compute the imports closure $\mathsf{clos}(\mathsf{G})$ of G by enriching G with directly and indirectly imported triples and then we transform $\mathsf{clos}(\mathsf{G})$ into $\mathsf{O_G}$ using the mapping process as defined in the OWL 2 Mapping to RDF Graphs specification. If the mapping fails, then G is not well-formed and, thus, cannot be used under the OWL 2 Direct Semantics.
2. We proceed similarly for G', obtaining $\mathsf{O_{G'}}$.
3. We check whether $\mathsf{O_G} \models \mathsf{O_{G'}}$, where $\models$ denotes the OWL Direct Semantics entailment relation. Most commonly OWL reasoners do this by searching for a countermodel, i.e., a model $\mathcal{I}$ that satisfies $\mathsf{O_G}$ and the negation of $\mathsf{O_{G'}}$. A problem is that not all axioms can be negated in OWL. Thus, it is usually required to reformulate the reasoning problem and deal with each axiom in $\mathsf{O_{G'}}$ separately.

We illustrate some of the problems that have to be addressed in an OWL DS entailment regime in Example 35 below.

Table 10. FSS version of the triples for Example 35

```
 (1')  Ontology(<http://example.org/myOntology>
 (2')    Declaration(ObjectProperty(:eats))
 (3')    Declaration(ObjectProperty(:contains))
 (4')    Declaration(Class(:Vegetarian))
 (5')    Declaration(Class(:Vegan))
 (6')    Declaration(Class(:MilkProduct))
 (7')    ClassAssertion(ObjectUnionOf(:Vegetarian :Vegan) :Birte)
 (8')    ObjectPropertyAssertion(:eats :Birte :Yoghurt)
 (9')    ObjectPropertyAssertion(:contains :Yoghurt :Milk)
(10')    ClassAssertion(:MilkProduct :Milk)
(11')    SubClassOf(ObjectSomeValuesFrom(:contains :MilkProduct) :MilkProduct)
(12')    SubClassOf(:Vegan ObjectAllValuesFrom(:eats ObjectComplementOf(:MilkProduct)))
       )
```

Example 35 We consider the query:

```
SELECT ?ind
WHERE { ?ind rdf:type :Vegetarian }
```

We assume that the default (and, hence, the active graph for the query) contains the triples from Table 9. Since the Direct Semantics is defined in terms of OWL structural objects, we first have to map the triples from Table 9 into OWL objects. The result of the mapping is shown in Table 10. Triple (1) results in the ontology header (1'). This triple does not contribute anything towards the logical consequences of the ontology, but is required to satisfy the constraints of OWL 2 DL. Similarly, Triples (2) to (6) result in the non-logical axioms (2') to (6'), which declare terms as classes or object properties. Such declarations are required to allow for an unambiguous parsing process. The remaining triples lead to logical axioms: Triple (7) is the same as in Example 33 and states that the individual :Birte is a vegan or a vegetarian.

Note that in the FSS version of (7') we have ObjectUnionOf whereas in the RDF triples, we just have unionOf. This is because the FSS makes it explicit whether the element is a class or a data range. In case of a data range DataUnionOf would be used. In order to be able to decide what applies, the declarations are used, e.g., from (4) and (5) (in FSS (4') and (5'), respectively), we know that :Vegetarian and :Vegan are classes. Triple (8) translates into an assertion saying that the individual :Birte :eats the individual :Yoghurt. In order to see whether this is a data or an object property assertion in the FSS, we can again use the declarations. Axiom (9') is obtained similarly. From (11), we obtain a more complicated axiom that states: if an element has a :contains relationship with something that is an instance of :MilkProduct, then this element is itself an instance of :MilkProduct. Finally, (12) translates into a statement that says that instances of the class :Vegan can only be related with the property :eats to something that is not an instance of :MilkProduct. For those more familiar with Description Logic syntax, Table 11 shows the logical axioms into Description Logic syntax with (7^*), (11^*), and (12^*) terminological (TBox) axioms and (8^*), (9^*), and (10^*) assertional (ABox) axioms.

In order to find the answers for the query under OWL DS entailment, we also need a version of the BGP that can be interpreted according to the OWL structural specification. One way of doing this would be to replace the variables with terms from the ontology, then map the resulting triples to OWL axioms, and check entailment. This would, however, require frequent parsing/mapping attempts that frequently will fail because we substituted a variable with a value

Table 11. Description Logic version of the logical axioms for Example 35

(7*)	$(\mathsf{Vegetarian} \sqcup \mathsf{Vegan})(\mathsf{Birte})$
(8*)	$\mathsf{eats}(\mathsf{Birte}, \mathsf{Yoghurt})$
(9*)	$\mathsf{contains}(\mathsf{Yoghurt}, \mathsf{Milk})$
(10*)	$\mathsf{MilkProduct}(\mathsf{Milk})$
(11*)	$\exists \mathsf{contains}.\mathsf{MilkProduct} \sqsubseteq \mathsf{MilkProduct}$
(12*)	$\mathsf{Vegan} \sqsubseteq \forall \mathsf{eats}.(\neg \mathsf{MilkProduct})$

that violates the OWL 2 DL constraints, e.g., when we replace the variable ?ind with a class name, e.g., :Vegan, we obtain a triple that cannot be mapped since :Vegan rdf:type :Vegetarian is not allowed in OWL 2 DL, i.e., rdf:type cannot be used to relate two classes. Since we know that :Vegetarian is a class from (4), we know that ?ind has to be instantiated with individual names. In order to avoid a parsing attempt for each possible assignment of variables, the choice has been made to extend OWL's structural specification to allow for variables in place of atomic objects such as individuals, classes, properties, or literals. We can then simply map a BGP into axioms from the extended specification. This yields:

ClassAssertion(:Vegetarian?ind)

For this axiom it is clear that ?ind occurs in an individual position and, therefore, has to be replaced with individual names from the queried ontology. For this example, we only have to substitute ?ind with :Birte. We could also use dedicated reasoner methods to retrieve instances of the class :Vegetarian without iterating over all individual names to obtain the query result:

?**ind**
:Birte

Note that the class used in the query pattern could equally be a class expression such as

ObjectUnionOf(:Vegetarian :Vegan ObjectAllValuesFrom(:eats:MilkProduct)),

although that last disjunct is somehow far-fetched as a class of things that only eat milk products. Assume further that we extend the ontology with:

(13) ClassAssertion(ObjectUnionOf(:Vegetarian :Vegan) :Ian)
(14) SubclassOf(:Vegetarian :HasSpecialMealRequest)
(15) SubclassOf(:Vegan :HasSpecialMealRequest)

Clearly :Ian belongs to the above stated disjunction, so should be returned as query answer although membership in that class is not explicitly stated nor can we foresee all such classes and extend the queried ontology accordingly. Furthermore, we might have to do case-based reasoning. In this case, we can neither extend the ontology with a statement that :Ian belongs to the class :Vegetarian nor with one that establishes that :Ian belongs to :Vegan. Nevertheless, we know that :Ian belongs to the extension of the class :HasSpecialMealRequest.

4.3 Mapping BGPs to Extended OWL Objects

Note that in the above example, it was clear from the queried ontology that ?ind rdf:type :Vegetarian corresponds to a class assertion with ?ind mapping to individual names since :Vegetarian was declared as a class in O_G. In some cases, however, the variables in a BGP do no longer allow for an unambiguous mapping, which is addressed by variable typing triples.

Variable Typing In order to have an unambiguous correspondence between BGPs and extended OWL objects, the Direct Semantics entailment regime requires for some cases extra triples in a basic graph pattern that give typing information for the variables.

Example 36 In order to see why this is required, consider the following query:

SELECT ?s ?p ?o WHERE { ?s ?p ?o }

Without any restrictions this query could be a query for

- declarations, i.e., the BGP maps to a declaration such as Declaration(Class(?s)) where ?p binds to rdf:type, ?o to owl:Class, and bindings for ?s have to be computed or Declaration(ObjectProperty(?s)) where ?p binds to rdf:type and ?o to owl:ObjectProperty, or any other type of declaration,
- inverse object properties, i.e., the BGP maps to ObjectInverseOf(?o) where ?s maps to a blank node and ?p to owl:inverseOf,
- subclasses, i.e., the BGP maps to SubClassOf(?s ?o) with rdfs:subClassOf as binding for ?p,
- equivalent classes, i.e., the BGP maps to EquivalentClasses(?s ?o) where ?p binds to owl:equivalentClass,
- disjoint classes, i.e., the BGP maps to DisjointClasses(?s ?o) where ?p binds to owl:disjointWith,
- ...

In order to answer the query without any typing constraints, all possible ways of mapping the BGP into ontology structures have to be considered. Even if variables can only occur in the position of function parameters of the functional-style syntax, the BGP from the above query can still be mapped to ObjectPropertyAssertion(?p ?s ?o), DataPropertyAssertion(?p ?s ?o), or AnnotationAssertion(?p ?s ?o) without variable typing information.

The inclusion of type declarations from the queried ontology means that at least the non-variable terms in the query can be disambiguated without additional typing information in the query. Typically, variables have to be declared if they represent classes, properties, or datatypes, whereas individual variables do not need declarations for an unambiguous mapping process. This is similar to typing in ontologies, where typing of individuals is optional, but typing for properties, classes, and non-OWL 2 datatypes is mandatory.

Table 12. Grammar extension for extended OWL objects

Class := IRI | Var	ObjectProperty := IRI | Var	DataProperty := IRI | Var
Individual := NamedIndividual | AnonymousIndividual | Var		
Literal := typedLiteral | stringLiteralNoLanguage | stringLiteralWithLanguage | Var		

Example 37 The BGP of the query

SELECT ?x WHERE { ?x :p ?y }

is parsed into (a) or (b) depending on whether :p is declared as an object or a data property in the queried ontology

(a) ObjectPropertyAssertion(:p ?x ?y) (b) DataPropertyAssertion(:p ?x ?y)

If :p is changed into the variable ?p, we need an extra typing triple, e.g.,

Declaration(ObjectProperty(?p))

to allow for an unambiguous mapping process.

Definition 11. *Let* BGP *be a basic graph pattern with* ?x *a variable occurring in* BGP*. If* BGP *contains a triple*

?x rdf:type TYPE,

where TYPE *is one of*

– owl:Class,
– owl:ObjectProperty,
– owl:DataProperty,
– owl:Datatype, *or*
– owl:NamedIndividual,

then ?x *is* declared *to be of type* TYPE.

From BGPs to Extended OWL Objects We now formally define how BGPs are mapped into OWL axioms extended to contain variables, i.e., the result of the mapping yields rather axiom *templates* than axioms.

The BGP of the query is mapped into an OWL 2 DL ontology, extended to allow variables in place of class names, object property names, datatype property names, individual names, or literals. Table 12 shows how productions of the OWL 2 functional-style syntax grammar [21] are extended to allow variables as defined by the Var production from the SPARQL grammar [26]. If BGP contains no ontology header, i.e., a triple of the form x rdf:type owl:Ontology with $x \in I \cup B$, we assume that BGP is extended with _:o rdf:type owl:Ontology for _:o a blank node name not occurring in BGP or the active graph before parsing BGP into extended OWL objects. Solution mappings in a query result are applied to such extended ontologies to obtain a set of OWL DL axioms that is compatible with the queried ontology and also entailed by it under DS.

Table 13. A query with infinitely many entailed solutions

```
G : :Peter a [ a owl:Restriction;                BGP : :Peter a [ a owl:Restriction;
      owl:onProperty :dp;                                  owl:onProperty :dp;
      owl:allValuesFrom [ a rdfs:Datatype;                 owl:allValuesFrom [ a rdfs:Datatype;
        owl:oneOf ("5"^^xsd:integer)]]                       owl:datatypeComplementOf [
                                                               a rdfs:Datatype; owl:oneOf (?x)]]]
```

O_G : ClassAssertion(DataAllValuesFrom(:dp DataOneOf("5"^^xsd:integer)) :Peter)
O^G_{BGP} : ClassAssertion(DataAllValuesFrom(:dp DataComplementOf(DataOneOf(?lit))) :Peter)

Definition 12. *An extended ontology* O^G_{BGP} *is constructed for a basic graph pattern* BGP *and graph* G *using the parsing process for RDF graphs as defined in [23] with three modifications:*

1. *variable identifiers are allowed in place of IRIs and literals in all parsing steps,*
2. *an ontology header may be added to* BGP *if not given, and*
3. *the type declarations given in* BGP *are augmented with the declarations in* G *and those obtained from graphs imported by* G *(denoted* AllDecl(G) *in [23]).*

The complete parsing process is detailed in the latest entailment regimes working draft.[5] *A basic graph pattern* BGP *satisfies the* typing constraints *of the entailment regime if*

– *no variable is declared as being of more than one type,*
– *variables without a type declaration occur either only in individual positions or only in literal positions, and*
– *it is possible to disambiguate all types of IRIs and variables when parsing* BGP *into extended OWL objects taking the typing information from* O_G *and from* BGP *into account;*

A basic graph pattern BGP *is* well-formed for the OWL DS entailment regime and a graph G *if* O^G_{BGP} *can be obtained in this way and is an extended OWL DL ontology. An RDF graph* G *is* well-formed for the OWL DS entailment regime *if is mapping to structural OWL objects [23], resulting in an ontology* O_G, *is defined.*

SPARQL Syntax Extensions for BGPs Considering the fact that each BGP has to be mapped to structural OWL objects anyway in order to use the OWL DS, it seems natural to directly allow for specifying BGPs in other OWL syntaxes, e.g., the FSS. Such an extension has not been specified by the W3C as part of the entailment regimes document, but it seems likely that implementations of the OWL DS regime might also accept other syntaxes for the BGP.

4.4 Infinite Entailments in Datatype Reasoning

[5] http://www.w3.org/TR/2010/WD-sparql11-entailment-20100601/

Example 38 In order to see how datatype reasoning in OWL can cause infinite entailments, consider the graph and query in Table 13. The graph G states that all data values to which Peter is related via :dp are in the singleton set of the integer 5. The BGP asks for all data values to which :Peter cannot be related with :dp. Without suitable restrictions, all (infinitely many) integers other than 5 could be used in solution mappings for ?x. Moreover, it is currently unknown how to compute all mappings for literal variables even for cases where there number is finite – testing all literals is clearly not an option.[6]

We will again use the vocabulary of the queried graph to include only literals that are explicitly mentioned in the input graph for the OWL entailment regimes. Like for the IRIs rdf:_i, this may lead to unexpected behavior, since mentioning a literal in the input may lead to new query results even for queries not directly related to this literal. Yet, this problem seems so rare in practice that a more detailed analysis of the problematic datatype expressions is not worthwhile, even if it could further limit unintuitive behavior.

4.5 The OWL 2 Direct Semantics Entailment Regime

We now define the evaluation of graph patterns. For the Direct Semantics, Skolemization is applied to O_G, which ensures that only blank nodes that represent anonymous OWL individuals are Skolemized, not blank nodes used for encoding complex OWL syntax in RDF.

Definition 13. *Let* G *be an RDF graph that is well-formed for the OWL 2 DS entailment regime,* BGP *a basic graph pattern that is well-formed for DS and* G, $\mathsf{V}(\mathsf{O}^\mathsf{G}_\mathsf{BGP})$ *the set of variables in* $\mathsf{O}^\mathsf{G}_\mathsf{BGP}$, $\mathsf{B}(\mathsf{O}^\mathsf{G}_\mathsf{BGP})$ *the set of blank nodes in* $\mathsf{O}^\mathsf{G}_\mathsf{BGP}$, sk *a Skolemization function for the blank nodes in* $\mathsf{O}^\mathsf{G}_\mathsf{BGP}$ *as in Definition 9 such that* $\mathsf{ran}(\mathsf{sk}) \cap (\mathsf{Voc}(\mathsf{O}_\mathsf{G}) \cup \mathsf{Voc}(\mathsf{O}^\mathsf{G}_\mathsf{BGP})) = \emptyset$.

We write $\models_\mathsf{DS}$ *for the OWL 2 Direct Semantics entailment relation and define the* evaluation of BGP over G under OWL 2 Direct Semantics entailment, $[\![\mathsf{BGP}]\!]^\mathsf{DS}_\mathsf{D,G}$, *as the solution multiset*

$\{(\mu, n) \mid \mathsf{dom}(\mu) = \mathsf{V}(\mathsf{BGP})$, *and n is the maximal number of distinct RDF instance mappings such that, for each* $1 \leq i \leq n$,
- *(i)* $\mathsf{dom}(\mathsf{sigma}_\mathsf{i}) = \mathsf{B}(\mathsf{BGP})$,
- *(ii)* $\mu(\sigma_i(\mathsf{O}^\mathsf{G}_\mathsf{BGP})) \cup \mathsf{O}^\mathsf{G}$ *is an OWL 2 DL ontology*,
- *(iii)* $\mathsf{sk}(\mu(\sigma_\mathsf{i}(\mathsf{O}^\mathsf{G}_\mathsf{BGP})))$ *are ground RDF triples*,
- *(iv)* $\mathsf{sk}(\mathsf{O}_\mathsf{G}) \models_\mathsf{DS} \mathsf{sk}(\mu(\sigma_\mathsf{i}(\mathsf{O}^\mathsf{G}_\mathsf{BGP})))$, *and*
- *(v)* $\mathsf{ran}(\mu) \subseteq \mathsf{Voc}(\mathsf{O}_\mathsf{G})\}$.

If O_G is inconsistent, queries must be rejected with an error.

Restrictions on Solutions Since solutions can only bind to terms from a finite vocabulary, clearly the solution multiset and each multiplicity is finite too. Although this avoids infinite results as discussed in Section 4.4, reasoners may have to consider a large number of literals as potential variable bindings and we expect that not all systems will provide a complete implementation for queries with literal variables.

Note that for the OWL DS regime no vocabulary other than that of the graph itself is required since there are no axiomatic triples and variables can only bind to built-in terms that are also built-in entities. Built-in entities such as owl:Thing are, however, assumed to be present in any ontology [21, Table 5], i.e., $\mathsf{O_G}$ automatically includes declarations for these built-in entities. Thus, we have omitted any OWL 2 specific vocabulary from condition(v).

Compared to the RDFS regime, condition (ii) requires $\mu(\sigma_i(\mathsf{O}^{\mathsf{G}}_{\mathsf{BGP}})) \cup \mathsf{O}^{\mathsf{G}}$ to be an OWL 2 DL ontology. Thus, the axioms from the instantiated BGP together with the axioms from the queried ontology must satisfy the restrictions for OWL 2 DL ontologies. These restrictions are in place to guarantee that the key reasoning tasks in OWL 2 with Direct Semantics are decidable. For example, for owl:topDataProperty, the following requirement has to be met in OWL 2 DL:

> The owl:topDataProperty property occurs in a SubDataPropertyOf axiom only in the position of the super-property.

The condition guarantees that these restrictions are equally applied to the query. Furthermore, the condition prevents that the BGP uses a property in a number restriction that is declared as transitive in the queried ontology since transitive properties cannot occur in number restrictions in OWL 2 DL.

The complexity of standard reasoning problems in OWL are well-understood and BGP evaluation can be implemented using the standard reasoning techniques. The complexity of OWL reasoning usually outweighs that of the SPARQL algebra operations, i.e., checking whether a solution mapping is a solution is complete for nondeterministic double exponential time in OWL 2 DL.

Higher Order Queries The Direct Semantics entailment regime allows for certain (but not all) forms of higher order queries.

Example 39 The BGP

?x rdfs:subClassOf ?y

can be used to query for pairs of sub- and super-classes. This means that variables can bind to classes (representing sets of individuals) and not just to individuals or data values.

Queries in which variables are used in positions of a First-Order Logic quantifier, will, however, be illegal since such queries cannot be mapped to OWL objects as required.

Example 40 The following (illegal) query asks whether some or all brothers of Peter are persons:

```
SELECT ?x
WHERE { :Peter a [
            a owl:Restriction ;
            owl:onProperty :hasBrother ;
            ?x :Person
        ]
      }
```

In FSS the BGP of the query corresponds to the axiom:

ClassAssertion(?x(:hasBrother :Person) :Peter)

Here the variable occurs in the position of a quantifier (ObjectSomeValuesFrom or ObjectAllValuesFrom, i.e., $\exists$ and $\forall$ in Description Logics) and not just in the position of OWL entities such as class names or individual names.

4.6 The OWL 2 RDF-Based Semantics Entailment Regime

The OWL 2 RDF-Based Semantics is a direct extension of the RDFS semantics, which means that it interprets RDF triples directly without the need of mapping an RDF graph into structural objects. Compared to the Direct Semantics, the RDF-Based Semantics treats classes as individuals that refer to elements of the domain. Each such element is then associated with a subset of the domain, called the class extension. This means that semantic conditions on class extensions are only applicable to those classes that are actually represented by an element of the domain which can lead to less consequences than expected. An example is given by the following graph and BGP:

```
G : :a rdf:type :C        BGP : ?x rdf:type [ rdf:type owl:Class ;
                                              owl:unionOf ( :C :D ) ]
```

G states that :a has type :C, while BGP asks for instances of the complex class denoting the union of :C and :D. One might expect μ: ?x $\mapsto$:a to be a solution, but this is not the case under the OWL 2 RDF-Based Semantics (see also [29, Sec. 7.1]). It is guaranteed that the union of the class extensions for :C and :D exists as a subset of the domain; no statement in G implies, however, that this union is the class extension of any domain element. Thus, μ(BGP) is not entailed by G.

The entailment holds, however, when the statement :E owl:unionOf (:C :D) is added to G. In the OWL Direct Semantics, in contrast, classes denote sets and not domain elements, so G entails μ(BGP) under DS where, formally, G must first be extended with an ontology header to become well-formed for DS. Note that a similar situation occurs for Example 38, but the problem of infinitely many answers occurs if the necessary expressions are introduced.

Summing up, the RBS handles blank nodes just like RDFS, even in cases where they are needed for encoding OWL class expressions. This allows us to use Skolemization just like in the case of RDFS in the next definition. The expressive datatype reasoning is again addressed as for the DS using the answer domain.

Definition 14. *Let* G *be an RDF graph,* BGP *a basic graph pattern,* $\mathsf{V(BGP)}$ *the set of variables in* BGP*,* $\mathsf{B(BGP)}$ *the set of blank nodes in* BGP*,* sk *a Skolemization function as in Definition 9 such that* $\mathsf{ran(sk)} \cap (\mathsf{Voc(G)} \cup \mathsf{Voc(BGP)}) = \emptyset$*. Let* $\mathsf{Voc(OWL2RB)}$ *be the OWL 2 RDF-Based vocabulary and* $\mathsf{Voc^{-}(OWL2RB)} = \mathsf{Voc(OWL2RB)} \setminus \{\mathsf{rdf{:}\_i} \mid i \in \mathbf{N}\}$.

We write $\models_{\mathsf{RBS}}$ *for the OWL 2 RDF-Based Semantics entailment relation and define the* evaluation of BGP over G under OWL 2 RDF-Based Semantics entailment*,* $[\![\mathsf{BGP}]\!]^{\mathsf{RBS}}_{\mathsf{D,G}}$*, as the solution multiset*

$\{(\mu, n) \mid \mathsf{dom}(\mu) = \mathsf{V(BGP)}$*, and n is the maximal number of distinct RDF instance mappings such that, for each* $1 \leq i \leq n$*,*
(i) $\mathsf{dom(sigma_i)} = \mathsf{B(BGP)}$,
(ii) $\mu(\sigma_i(\mathsf{BGP}))$ *are well-formed RDF triples,*
(iii) $\mathsf{sk}(\mu(\sigma_i(\mathsf{BGP})))$ *are ground RDF triples,*
(iv) $\mathsf{sk(G)} \models_{\mathsf{RBS}} \mathsf{sk}(\mu(\sigma_i(\mathsf{BGP})))$*, and*
(v) $\mathsf{ran}(\mu) \subseteq \mathsf{Voc(G)} \cup \mathsf{Voc^{-}(OWL2RB)}\}$.

4.7 OWL 2 Profiles

OWL 2 DL is decidable, but computationally hard and not scalable enough for many applications. OWL Full is not even decidable and, consequently, not many implementations that support all of OWL Full are available. Thus, OWL 2 identifies subsets of OWL 2, called *profiles*, which are sufficiently expressive, but of lower complexity (tractable) and tailored to specific reasoning services (see also Figure 1):

– Terminological/schema reasoning: OWL 2 EL
– Query Answering via database engines: OWL 2 QL
– Assertional/data reasoning with rule engines: OWL 2 RL

The OWL 2 QL and EL profiles further restrict the allowed inputs compared to OWL 2 DL, but equally use the Direct Semantics. The OWL 2 RL profile, in principle, can be used with both semantics, but for the Direct Semantics the input RDF graph has to satisfy some constrains. The RDF-Based semantics can be use with any RDF graph but under the OWL 2 RL profile one derives only certain consequences.

OWL 2 DL is the largest subset of RDF graphs for which the OWL 2 Direct Semantics is defined. Systems that support OWL 2 DL can also handle ontologies that satisfy the restrictions of the OWL 2 EL and QL profiles because these profiles are even more restrictive.

The OWL 2 EL Profile is particularly useful in applications employing ontologies that contain very large numbers of properties and/or classes. The profile captures the expressive power used by many ontologies and is a subset of OWL 2 DL for which the basic reasoning problems can be performed in time that is polynomial with respect to the size of the ontology. Worth mentioning is that the class hierarchy (all subclass relations between classes) can be computed in "one pass", whereas OWL 2 DL reasoner

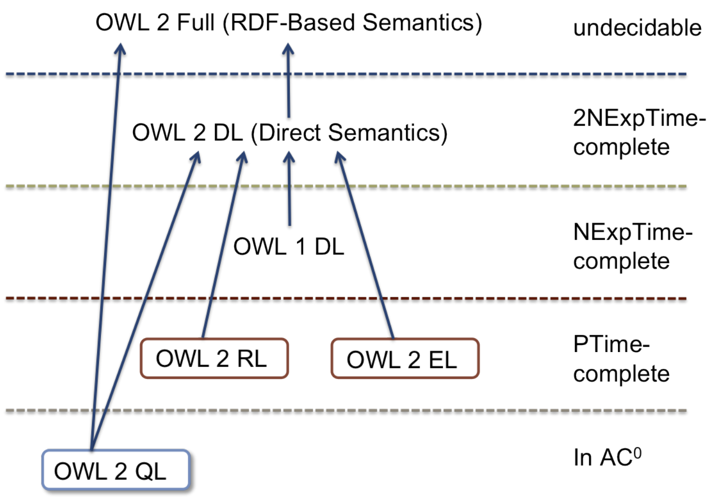

Fig. 1. An overview for the complexity of reasoning in OWL and its profiles

typically have to check each pair of classes separately. The one-pass classification exploits saturation-based techniques developed for $\mathcal{EL}$ Description Logics [2, 1, 7, 3, 7] and can be extended to the Horn (non-disjunctive) fragment of OWL DL [16].

The OWL 2 QL Profile is aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task. In OWL 2 QL, conjunctive query answering can be implemented using conventional relational database systems [9, 10, 25]. Using query rewriting techniques, sound and complete conjunctive query answering can be performed in LogSpace with respect to the size of the data (assertions) using standard database management systems. Recently developed techniques prevent an exponential blowup from query rewriting [18, 27]. As in OWL 2 EL, polynomial time algorithms can be used to implement the ontology consistency and class expression subsumption reasoning problems.

Note that OWL 2 QL implementations most commonly will only support conjunctive queries, i.e., queries where the BGP consists only of axioms of the following type:

– ClassAssertion,
– ObjectPropertyAssertion, and
– DataPropertyAssertion.

With the additional restriction that variables can only occur in the position of individuals and literals (if datatype reasoning is supported). Future versions of SPARQL could define further entailment regimes, e.g., one that defines a dedicated conjunctive query regime. Since an implementations is, however, free to reject any unsupported query anyway, the currently defined OWL regime can still be used.

The OWL 2 RL Profile defines a syntactic subset of OWL 2, which is amenable to implementation using rule-based technologies. For RDF graphs that fall into this syntactic subset, reasoning is sound and complete and both semantics of OWL can be used yielding the same results. Outside of this syntactic fragment, the RDF-Based Semantics can still be used, but reasoning can be incomplete. The main reasoning in the RL profile are PTime-complete (ontology consistency, class expression satisfiability, class expression subsumption, instance checking, and conjunctive query answering). Reasoning can be implemented in a rule engine (with equality support) by materializing schema inferences for facts.

4.8 Implementing the OWL 2 RL Profile via Rules

The OWL 2 RL specification provides a complete rule set that can be used to materialize all OWL 2 RL inferences. Each RDF triple is encoded via a ternary predicate T(_, _, _). A given set of rules is then applied to the ternary predicates.

Example 41 Subproperty reasoning is, for example, handled via the rule `prp-spo1`:

`prp-spo1`: $T(?p_1, \text{rdfs:subPropertyOf}, ?p_2) \land T(?x, ?p_1, ?y) \rightarrow T(?x, ?p_2, ?y)$

Given the first two triples below (as ternary predicates), we can derive the third one by applying the above rule:

T(:hasSister, rdfs:subPropertyOf, :hasSibling)
T(:Peter, :hasSister, :Mary)

⇒ T(:Peter, :hasSibling, :Mary)

Functionality for properties is taken into account via the `prp-fp` rule:

`prp-fp`: $T(?p, \text{rdf:type}, \text{owl:FunctionalProperty}) \land T(?x, ?p, ?y_1) \land T(?x, ?p, ?y_2)$
$\rightarrow T(?y_1, \text{owl:sameAs}, ?y_2)$

Given the first three triples, we can then apply the rule to derive the forth triple:

T(:hasMother, rdf:type, owl:FunctionalProperty)
T(:John, :hasMother, :Anna)
T(:John, :hasMother, :Ann)

⇒ T(:Anna, owl:sameAs, :Ann)

We illustrate how subclass reasoning with complex class expressions can be performed using the data from Table 14.

`cax-sco`: $T(?c_1, \text{rdfs:subClassOf}, ?c_2) \land T(?x, \text{rdf:type}, ?c_1) \rightarrow T(?x, \text{rdf:type}, ?c2)$
`cls-avf`: $T(?x, \text{owl:allValuesFrom}, ?y) \land T(?x, \text{owl:onProperty}, ?p) \land$
$T(?u, \text{rdf:type}, ?x) \land T(?u, ?p, ?v) \rightarrow T(?v, \text{rdf:type}, ?y)$

The rule `cax-sco` can be applied to the ternary form of triple (1a) and (3) to derive the first of the two triples below. Then, the ternary form of triples (1c), (1d), (4), and (2) can be used to satisfy the body of the rule `cls-avf` binding ?x to _:c, ?y to :Person, ?p to :hasChild, ?u to :Anna, and ?v to :Mary, to derive triple (5).

Table 14. Data used to illustrate subclass reasoning with complex class expressions in OWL RL

Turtle:	(1a) :Person rdfs:subClassOf _:c
	(1b) _:c rdf:type owl:Restriction
	(1c) _:c owl:allValuesFrom :Person
	(1d) _:c owl:onProperty :hasChild
	(2) :Anna :hasChild :Mary
	(3) :Anna rdf:type :Person
FSS:	(1) SubClassOf(:Person ObjectAllValuesFrom(:hasChild :Person))
	(2) ObjectPropertyAssertion(:hasChild :Anna :Mary)
	(3) ClassAssertion(:Person :Anna)
DL:	(1) $\mathsf{Person} \sqsubseteq \forall \mathsf{hasChild.Person}$
	(2) hasChild(Anna, Mary)
	(3) Person(Mary)

⇒ (4) :Anna rdf:type _:c
⇒ (5) :Mary rdf:type :Person

Note that triple (4) has no representation in FSS or DL notation and would not be derived by a non-rule-based OWL reasoner that uses the Direct Semantics. The triple is rather an intermediate consequence with the purpose of deriving the class assertion (5).

After exhaustively applying the OWL RL rules [20] to a set of RDF triples, the resulting extended graph contains triples that state the (atomic) types for all individuals as well as the relationships between individuals. Schema reasoning is, however, not performed by applying the OWL 2 RL rules, i.e., we do not have all triples $:c_1$ rdfs:subClassOf $:c_2$ for $:c_1$ a subclass of $:c_2$ under the Direct or RDF-Based semantics.

In order to evaluate BGP over an active graph G using the OWL 2 RL profile one can proceed as follows:

1. Saturate G using the OWL 2 RL rule to obtain G'.
2. Evaluate BGP over G' using sub-graph matching (i.e., via any standard SPARQL implementation).

More optimized implementation than via the fixed OWL 2 RL rule set are possible [19]. It is further possible to implement the RL profile in any rule engine that supports the RIF Core dialect [6, 8] either as fixed or ontology-specific rule set.

5 Exercises

We provide a couple of exercises in this section that can be used to test the understanding of several aspects that have been presented in the previous sections. Solutions to the exercises are provided in the following section.

5.1 Mapping to the SPARQL Algebra

Exercise 1 *Translate the following SPARQL query into an abstract query:*

```
SELECT ?mbox
WHERE { ?x foaf:mbox ?mbox }
```

Exercise 2 *Translate the following SPARQL query into an abstract query:*

```
SELECT DISTINCT ?name
WHERE { ?x foaf:name ?name FILTER regex(?name, ”ian”) }
```

Exercise 3 *Translate the following SPARQL query into an abstract query:*

```
SELECT ?mbox
WHERE { { ?x foaf:name "Birte Glimm". ?x foaf:mbox ?mbox }
           UNION
         { ?x foaf:name ?name . ?x foaf:mbox ?mbox
            FILTER regex(?name, ”ian”) }
       }
```

Exercise 4 *Translate the following SPARQL query into an abstract query:*

```
SELECT ?name ?id
WHERE { { ?x foaf:name ?name OPTIONAL { ?x foaf:icqChatID ?id } }
          UNION { ?x foaf:name ?name . ?x foaf:mbox <mailto:rudolph@kit.edu> }
      } ORDER BY ?name
```

5.2 Query Evaluation

For the query evaluation in this section we assume simple entailment, i.e., subgraph matching.

Exercise 5 *Illustrate the evaluation of the query from Exercise 3 including its intermediate results assuming the default graph contains the triples from Table 1.*

Exercise 6 *Illustrate the evaluation of the query from Exercise 4 including its intermediate results assuming the default graph contains the triples from Table 1.*

5.3 RDFS Semantics Queries

In this section we assume RDFS entailment, i.e., we use the RDFS entailment regime.

Table 15. RDF triples for Exercise 7

```
     @prefix  : <http://example.org/> .
     @prefix  w3c: <http://www.w3.org/> .
     @prefix  iswc2010: <http://data.semanticweb.org/conference/iswc/2010/> .
 (1) iswc2010:paper/280        rdf:type              :ConferencePaper.
 (2) iswc2010:paper/280        :authors              _:l1.
 (3) _:l1                      rdf:type              rdf:Seq.
 (4) _:l1                      rdf:_1                "Birte Glimm".
 (5) _:l1                      rdf:_2                "Markus Krötzsch".
 (6) w3c:TR/rdf-sparql-query   rdf:type              :W3CStandard.
 (7) w3c:TR/rdf-sparql-query   :writtenBy            _:l2.
 (8) _:l2                      rdf:type              rdf:Seq.
 (9) _:l2                      rdf:_1                "Eric Prud'hommeaux".
(10) _:l2                      rdf:_2                "Andy Seaborne".
(11) :ConferencePaper          rdfs:subClassOf       :Publication.
(12) :W3CStandard              rdfs:subClassOf       :Publication.
(13) :writtenBy                rdfs:subPropertyOf    :authors.
```

Exercise 7 *We assume a graph with the triples from Table 15 and the query:*

```
SELECT ?auth ?pub
WHERE { ?pub rdf:type :Publication . ?pub :authors ?seq . ?seq ?ind ?auth }
```

List the query results under the RDFS entailment regime and argue, for each solution, why the solution follows.

Exercise 8 *You might have noticed that the query from Exercise 7 has two answers in which the binding for* ?auth *is not an author name. How can we modify the query to query for solutions in which* ?auth *binds to an author name?*

Exercise 9 *We again assume a graph with the triples from Table 15 and the query:*

```
SELECT ?type
WHERE { iswc2010:paper/280 rdf:type ?type }
```

Which answers does the query have under RDFS entailment and why?

Exercise 10 *We again assume a graph with the triples from Table 15. Is the triple* iswc2010:paper/280 :authors _:x *entailed under RDFS entailment? What is then the answer to the following query?*

```
ASK { iswc2010:paper/280 :authors _:x }
```

5.4 OWL Direct Semantics Queries

Exercise 11 *We assume that the queried ontology contains the axioms from Table 9. Map the following BGP into an extended OWL axiom, list the results of evaluating the BGP under OWL Direct Semantics, and explain, for each solution, why the solution is entailed:*

?mp rdf:type :MilkProduct

Exercise 12 *Map the query pattern of the following query into extended OWL objects and illustrate the evaluation of the query over the ontology from Table 9:*

```
SELECT ?sup
WHERE { :MilkProduct rdfs:subClassOf ?sup. ?sup rdf:type owl:Class }
```

Exercise 13 *We consider the ontology from Table 9. Why is the query*

```
SELECT ?rel
WHERE { :Vegetarian ?rel :Vegan }
```

not a well-formed query under the OWL 2 Direct Semantics?

Exercise 14 *What query can one use to retrieve a list of all classes tat occur in the ontology?*

Exercise 15 *A typical reasoning tasks in OWL is the classification of classes, i.e., the computation of all pairs $\langle C, D\rangle$ such that C is a direct sub-class of D or C is equivalent to D. Can a SPARQL query be used to retrieve the subsumption hierarchy?*

Exercise 16 *Can the OWL Direct Semantics entailment regime be implemented via materialization, as sketched for the RDFS regime? If so, sketch what one would have to do. If not, why is it no possible and would it possible for subsets of the language?*

6 Solutions to the Exercises

In this section, we provide the solution for the exercises from the previous section.

6.1 Mapping to the SPARQL Algebra

Solution 1 We start with the query pattern, which is, as every query pattern, a group graph pattern here consisting of one element, which is a `TriplesBlock`. Since Definition 3 defines the translation for `GroupGraphPattern` according to Algorithm 1 (we have no filter, but one other element, which is the BGP), we get Join(Z, algbr(bgp)) with bgp the BGP of the query.

Going back to Definition 3 for the translation of the BGP, we can now use the first case for `TriplesBlock` and we obtain Join(Z, Bgp(?x foaf:mbox ?mbox)). The object can be simplified to just Bgp(?x foaf:mbox ?mbox).

Now that we have the algebra translation for the query pattern, which we denote with E, we can obtain the algebra translation for the whole query and then the abstract query as described in Definition 8. We first obtain ToList(E), then go on to Project(ToList(E), {?mbox}). Finally, we obtain the abstract query (assuming D is the dataset):

(Project(ToList(E), {?mbox}), D, SELECT)

Solution 2 We start again with the query pattern, which is again a group graph pattern this time consisting of an element (a `TriplesBlock`) with a filter. We translate according to Algorithm 1 and then according to the case for `GroupGraphPattern` with one filter and one element. We have to apply TranslateGroup and obtain, as in the previous exercise, Join(Z, Bgp(?x foaf:name ?name)), which we simplify to Bgp(?x foaf:name ?name). Together with the filter translation, this results in

Filter(regex(?name, ”ian”), Bgp(?x foaf:name ?name)).

Now that we have the algebra translation for the query pattern, which we denote with E, we can obtain the algebra translation for the whole query according to Definition 8. After applying ToList(E) and Project(ToList(E), {?name}) as above, we further translate the DISTINCT keyword and obtain:

```
(Distinct(Project( ToList(Filter(regex(?name, ”ian”), Bgp(?x foaf:name ?name )))
                   {?mbox})), D, SELECT)
```

Solution 3 We start with the query pattern, which is, as every query pattern, a group graph pattern consisting of one element, which is a `GroupOrUnionGraphPattern` of the form

`GroupGraphPattern` UNION `GroupGraphPattern`

as can be seen from the grammar in Table 2. Thus, we start with a translation according to Algorithm 1 and then according to the case for `GroupOrUnionGraphPattern` from Definition 3 obtaining: Join(Z, Union(algbr(G_1), algbr(G_2))) with G_1 andG_2 denoting the first and the second group of the union, respectively. For G_1 we again use Algorithm 1 followed by the case for `TriplesBlock` from Definition 3, leading to

Join(Z, Bgp(?x foaf:name "Birte Glimm" . ?x foaf:mbox ?mbox)).

Since G_2 has a filter, we obtain

```
Filter( regex(?name, ”ian”),
        Join(Z, Bgp(?x foaf:name ?name . ?x foaf:mbox ?mbox))).
```

Putting all together, we get:

```
Join(Z, Union( Join(Z, Bgp(?x foaf:name "Birte Glimm" . ?x foaf:mbox ?mbox)),
               Filter( regex(?name, ”ian”),
                       Join(Z, Bgp(?x foaf:name ?name . ?x foaf:mbox ?mbox)))))
```

which can be simplified to

$$\begin{aligned}&\mathsf{Union(\ Bgp(?x\ foaf{:}name\ "Birte\ Glimm"\ .\ ?x\ foaf{:}mbox\ ?mbox),}\\ &\qquad\mathsf{Filter(regex(?name,\ "ian"),\ Bgp(?x\ foaf{:}name\ ?name\ .\ ?x\ foaf{:}mbox\ ?mbox)))}\end{aligned}$$

Now that we have the algebra translation for the query pattern, which we denote with E, we can obtain the algebra translation for the whole query and then the abstract query as described in Definition 8:

$$(\mathsf{Project(ToList(E)}, \{\mathsf{?mbox}\}), \mathsf{D}, \mathsf{SELECT})$$

Solution 4 We again translate the query pattern first obtaining:

$$\begin{aligned}&\mathsf{Union(\ LeftJoin(Join(Z,\ Bgp(?x\ foaf{:}name\ ?name)), Bgp(?x\ foaf{:}icqChatID\ ?id)}, \texttt{true}),\\ &\qquad\mathsf{Bgp(?x\ foaf{:}name\ ?name\ .\ ?x\ foaf{:}mbox\ <mailto{:}rudolph@kit.edu>))}\end{aligned}$$

The expression can be simplified to:

$$\begin{aligned}&\mathsf{Union(LeftJoin(Bgp(?x\ foaf{:}name\ ?name), Bgp(?x\ foaf{:}icqChatID\ ?id)}, \texttt{true}),\\ &\qquad\mathsf{Bgp(?x\ foaf{:}name\ ?name\ .\ ?x\ foaf{:}mbox\ <mailto{:}rudolph@kit.edu>))}\end{aligned}$$

We refer to the simplified expression as E and obtain the abstract query:

$$(\mathsf{Project(OrderBy(ToList(E)}, (\mathsf{ASC(?name)})), \{\mathsf{?name}, \mathsf{?id}\}))$$

6.2 Query Evaluation

Solution 5 We evaluate the algebra expression inside out, starting with the BGPs. The evaluation of Bgp(?x foaf:name "Birte Glimm" . ?x foaf:mbox ?mbox) yields $\Omega_1 = \{\mu_1\}$ with

$$\mu_1 : \mathsf{?x} \mapsto \mathsf{\_{:}a}, \mathsf{?mbox} \mapsto \mathsf{"b.glimm@googlemail.com"}.$$

The evaluation of Bgp(?x foaf:name ?name . ?x foaf:mbox ?mbox) yields $\Omega_2 = \{\mu_2, \mu_3\}$ with

$$\begin{aligned}\mu_2 &: \mathsf{?x} \mapsto \mathsf{\_{:}a}, \mathsf{?name} \mapsto \mathsf{"Birte\ Glimm"}, \mathsf{?mbox} \mapsto \mathsf{"b.glimm@googlemail.com"},\\ \mu_3 &: \mathsf{?x} \mapsto \mathsf{\_{:}b}, \mathsf{?name} \mapsto \mathsf{"Sebastian\ Rudolph"}, \mathsf{?mbox} \mapsto \mathsf{<mailto{:}rudolph@kit.edu>}.\end{aligned}$$

We next evaluate Filter(regex(?name, "ian"), Ω_2) obtaining $\Omega'_2 = \{\mu_3\}$. We can now evaluate the union operator, which yields $\Omega = \{\mu_1, \mu_3\}$, which is then turned into a list by the ToList operator. Applying the projection operator yields the final solution sequence: (μ'_1, μ'_3) with

$$\begin{aligned}\mu'_1 &: \mathsf{?mbox} \mapsto \mathsf{"b.glimm@googlemail.com"},\\ \mu'_3 &: \mathsf{?mbox} \mapsto \mathsf{<mailto{:}rudolph@kit.edu>}.\end{aligned}$$

Solution 6 We again evaluate the algebra expression inside out, starting with the BGPs. The evaluation of Bgp(?x foaf:name ?name) yields $\Omega_1 = \{\mu_1^1, \mu_1^2, \mu_1^3\}$ with

$$\begin{aligned}\mu_1^1 &: \mathsf{?x} \mapsto \mathsf{\_{:}a}, \mathsf{?name} \mapsto \mathsf{"Birte\ Glimm"},\\ \mu_1^2 &: \mathsf{?x} \mapsto \mathsf{\_{:}b}, \mathsf{?name} \mapsto \mathsf{"Sebastian\ Rudolph"},\\ \mu_1^3 &: \mathsf{?x} \mapsto \mathsf{\_{:}c}, \mathsf{?name} \mapsto \mathsf{"Pascal\ Hitzler"}.\end{aligned}$$

The evaluation of Bgp(?x foaf:icqChatID ?id) yields $\Omega_2 = \{\mu_2^1\}$ with

$$\mu_2^1 : \mathsf{?x} \mapsto \mathsf{\_{:}a}, \mathsf{?id} \mapsto \mathsf{"b.glimm"}.$$

For Bgp(?x foaf:name ?name . ?x foaf:mbox <mailto:rudolph@kit.edu>) we obtain $\Omega_3 = \{\mu_3^1\}$ with

$$\mu_3^1\colon \text{?x} \mapsto \text{\_:b}, \text{?name} \mapsto \text{"Sebastian Rudolph"}.$$

In order to evaluate LeftJoin(Ω_1, Ω_2, true), we first compute Filter(true, Join(Ω_1, Ω_2)) which yields $\Omega_4 = \{\mu_4^1\}$ with

$$\mu_4^1\colon \text{?x} \mapsto \text{\_:a}, \text{?name} \mapsto \text{"Birte Glimm"}, \text{?id} \mapsto \text{"b.glimm"}.$$

The mappings μ_1^2 and μ_1^3 cannot be joined with μ_2^1 since they are not compatible. Due to the incompatibility, both these mapping participate, however, in the union and are part of the solution for LeftJoin(Ω_1, Ω_2, true) due to the second part of the LeftJoin definition. Evaluating LeftJoin(Ω_1, Ω_2, true) yields $\Omega_5 = \{\mu_4^1\} \cup \{\mu_1^2, \mu_1^3\} = \{\mu_5^1, \mu_5^2, \mu_5^3\}$ with

$$\begin{aligned}
\mu_5^1 = \mu_4^1 &: \text{?x} \mapsto \text{\_:a}, \text{?name} \mapsto \text{"Birte Glimm"}, \text{?id} \mapsto \text{"b.glimm"},\\
\mu_5^2 = \mu_1^2 &: \text{?x} \mapsto \text{\_:b}, \text{?name} \mapsto \text{"Sebastian Rudolph"},\\
\mu_5^3 = \mu_1^3 &: \text{?x} \mapsto \text{\_:c}, \text{?name} \mapsto \text{"Pascal Hitzler"}.
\end{aligned}$$

We can now evaluate Union(Ω_5, Ω_3), which yields $\Omega_6 = \{\mu_6^1, \mu_6^2, \mu_6^3, \mu_6^4\}$ with

$$\begin{aligned}
\mu_6^1 &: \text{?x} \mapsto \text{\_:a}, \text{?name} \mapsto \text{"Birte Glimm"}, \text{?id} \mapsto \text{"b.glimm"},\\
\mu_6^2 &: \text{?x} \mapsto \text{\_:b}, \text{?name} \mapsto \text{"Sebastian Rudolph"},\\
\mu_6^3 &: \text{?x} \mapsto \text{\_:c}, \text{?name} \mapsto \text{"Pascal Hitzler"},\\
\mu_6^4 &: \text{?x} \mapsto \text{\_:b}, \text{?name} \mapsto \text{"Sebastian Rudolph"}
\end{aligned}$$

The multiset Ω_6 is then turned into a list by the ToList operator. Applying the OrderBy operator yields the list $(\mu_6^1, \mu_6^3, \mu_6^2, \mu_6^4)$. Finally, applying the projection operator yields: $(\mu_7^1, \mu_7^2, \mu_7^3, \mu_7^4)$ with

$$\begin{aligned}
\mu_7^1 &: \text{?name} \mapsto \text{"Birte Glimm"}, \text{?id} \mapsto \text{"b.glimm"},\\
\mu_7^2 &: \text{?name} \mapsto \text{"Pascal Hitzler"},\\
\mu_7^3 &: \text{?name} \mapsto \text{"Sebastian Rudolph"},\\
\mu_7^4 &: \text{?name} \mapsto \text{"Sebastian Rudolph"}.
\end{aligned}$$

6.3 RDFS Semantics Queries

Solution 7 We first list triples that are entailed under RDF semantics that are contributing solutions. The entailment follows from the RDFS entailment rules [14]. The relevant rule and the triples to which the rule is applied are indicated in the left-hand side column.

rdfs9 + (1) + (11) →	(14)	iswc2010:paper/280	rdf:type	:Publication.
rdfs9 + (6) + (11) →	(15)	w3c:TR/rdf-sparql-query	rdf:type	:Publication.
rdfs7 + (7) + (13) →	(15)	w3c:TR/rdf-sparql-query	:authors	_:l2.

If we were to materialize all RDFS-entailed triples, there would be several additional triples, but we focus here on the relevant ones. Although the above RDFS-entailed triples do not contain freshly generated blank nodes, we want to point out that sometimes blank nodes have to be introduced in the rule application process, but such freshly introduced blank nodes cannot be returned in a solution since they are not part of the answer domain. We obtain the following solutions from evaluating the BGP:

	?pub	?seq	?ind	?auth
μ_1 :	iswc2010:paper/280	_:l1	rdf:type	rdf:Seq
μ_2 :	iswc2010:paper/280	_:l1	rdf:_1	"Birte Glimm"
μ_3 :	iswc2010:paper/280	_:l1	rdf:_2	"Markus Krötzsch"
μ_4 :	w3c:rdf-sparql-query	_:l2	rdf:type	rdf:Seq
μ_5 :	w3c:rdf-sparql-query	_:l2	rdf:_1	"Andy Seaborne"
μ_6 :	w3c:rdf-sparql-query	_:l2	rdf:_2	"Eric Prud'hommeaux"

Computing the projection is then straightforward.

Solution 8 One possibility would be to apply a filter to ?auth that only permits literals as binding:

```
SELECT ?auth ?pub
WHERE { ?pub rdf:type :Publication . ?pub :authors ?seq . ?seq ?ind ?auth
          FILTER isLiteral(?auth) }
```

Other solutions with different filters are equally possible.

Solution 9 We first list triples that are entailed under RDF semantics that are contributing solutions. The entailment follows from the RDFS entailment rules [14]. The relevant rule and the triples to which the rule is applied are indicated in the left-hand side column.

rdfs4a + (1) →	(14)	iswc2010:paper/280 rdf:type rdfs:Resource.
rdfs9 + (1) + (11) →	(15)	iswc2010:paper/280 rdf:type :Publication.

thus, the query has two answers. The first inference might be surprising, but under RDFS entailment, we derive several such triples. If such triples are not desired, a filter can again be used to filter them out.

Solution 10 The triple iswc2010:paper/280 :authors _:x is indeed entailed under RDFS semantics since entailment treats black nodes as existential variables. According to triple (2), iswc2010:paper/280 is related via the property :authors to *some* element, witnessed by the blank node _:l1 in the data. Since the actual names of variables do not matter, i.e., the only question to decide is whether there is some element such that iswc2010:paper/280 is related to this element with the property :authors, which is the case.

Regarding the Boolean query (here we only have a blank node, no variable), we have two possible outcomes: there is a solution sequence containing a mapping (μ) where μ has an empty domain (it does not map any variable to anything) or there is only an empty solution sequence ( ). In the first case, the query answer is yes (true), whereas in the second case the query answer is no (false).

For the RDFS entailment regime, we work with a Skolem function that maps blank nodes from the active graph to constants, i.e., to fresh terms that occur neither in the query nor in the active graph. Let us assume that _:l1 is mapped to sk(l1). Since the query contains a blank node, we have to find an RDF instance mapping such that when we apply the mapping and then use the same Skolem function, the triples are entailed and ground. Thus, let μ be the mapping with an empty domain and σ: _:x $\mapsto$ _:l1, then

$$\mu(\sigma(\text{iswc2010:paper/280 :authors \_:x})) = \text{iswc2010:paper/280 :authors sk(l1)},$$

which is a ground triple that is entailed by sk(G) (even contained in sk(G)). Thus, the query answer is true.

6.4 OWL Direct Semantics Queries

Solution 11 The BGP is mapped into

FSS: ClassAssertion(:MilkProduct ?mp) DL: MilkProduct(?mp)

using the declaration axiom (6). Evaluating the BGP yields two solutions:

$$\begin{aligned}\mu_1 &: \text{?mp} \mapsto \text{:Yoghurt}\\ \mu_2 &: \text{?mp} \mapsto \text{:Milk}\end{aligned}$$

where μ_2 is a direct consequence of Axiom (10) and μ_1 follows from Axiom (9) and (11).

Solution 12 The BGP of the query pattern is mapped into SubClassOf(:MilkProduct ?sup) (DL: MilkProduct $\sqsubseteq$?sup). Evaluating the mapped BGP yields $\Omega = \{\mu_1, \mu_2\}$ with

$$\begin{aligned}\mu_1 &: \text{?sup} \mapsto \text{:MilkProduct}\\ \mu_2 &: \text{?mp} \mapsto \text{owl:Thing.}\end{aligned}$$

The solution μ_1 follows since each class is a subclass of itself under the DS (the subclass relation is reflexive) and μ_2 follows since owl:Thing is a superclass of every class. Applying ToList and Project yields the solution sequence (μ_1, μ_2).

Solution 13 The first problem is that ?rel is not typed. This makes it difficult to map the query pattern into an extended OWL object. Even worse, no matter what type we could add, the query cannot be fixed. Two classes, such as :Vegetarian and :Vegan, can only be related with terms from the special vocabulary, e.g., by saying that :Vegan is a subclass of :Vegetarian (in Turtle: :Vegan rdfs:subClassOf :Vegetarian) or by saying that the two classes are disjoint (in Turtle: :Vegetarian owl:disjointWith :Vegan). However, since terms of the special vocabulary do not have any of the types that variables can take, the query pattern cannot be fixed.

Solution 14 The query

```
SELECT ?class
WHERE { ?class rdfs:subClassOf owl:Thing . ?class rdf:type owl:Class }
```

would retrieve all classes of the ontology since any class is a subclass of owl:Thing (in DL: $\top$) under OWL's semantics. The typing triple is not necessary in this case since the parsing is unambiguous given that owl:Thing assumed to be declared as a class in any ontology even if such a declaration is not explicitly present.

Solution 15 A SPARQL query cannot distinguish between direct and indirect subclasses. Thus, a single query can, in general, not be used to retrieve all and only the required pairs. One would also get the indirect subclasses and it would be difficult to filter them out, at least in a single query.

Solution 16 If completeness is required, i.e., we want to return all solutions that are solutions, then materialization cannot be used as a general implementation technique. One of the problems are disjunctions, i.e., there is not just one canonical model of an OWL ontology that represents all relevant possible states of the world. One could argue that we could just include facts that hold in every model, e.g., if we have

```
:a rdf:type :C .
:b rdf:type [ rdf:type owl:Class ; owl:unionOf ( :D :E :F ) ].
:F rdfs:subClassOf owl:Nothing .
```

which is

$$\begin{array}{l} C(a) \\ (D \sqcup E \sqcup F)(b) \\ F \sqsubseteq \bot \end{array}$$

in DL notation, then we could argue that we add

```
:b rdf:type [ rdf:type owl:Class ; owl:unionOf ( :D :E ) ]
```

which is

$$(D \sqcup E)(b)$$

in DL notation to obtain a “canonical” model (since :F is a subclass of owl:Nothing it cannot have any instances). However, a BGP such as

```
?ind rdf:type [ rdf:type owl:Class ; owl:unionOf ( :C :D :F ) ]
```

would still have :a and :b as solutions (:a since it belongs to :C and :b as it belongs to the union of :D and :E). It would be impossible to foresee all such queries and materialize the required axioms in a finite ontology.

This is different for the OWL RL profile. The semantics of OWL 2 RL is defined such that certain consequences have to be derived, e.g., one materializes only (named) classes to which an individual belongs. The OWL 2 RL specification includes a set of rules that materialize all such consequences. Under certain restrictions for the ontology, the OWL RL rules derive all consequences that one would derive under the Direct Semantics. If the ontology violates the restrictions, then one might miss some answers that a tool that implements OWL 2 with its Direct Semantics could derive.

7 Links and Further Reading

The following list of references is not meant to be complete and is a subjective selection by the author. References that are not listed can equally be relevant and students are encouraged to look for references that most closely fit with their interests.

A text book covering the topics relevant for this summer school is: *Foundations of Semantic Web Technologies* Hitzler, P., Krötzsch, M., Rudolph, S. CRC Press 2009

7.1 Public SPARQL Endpoints

data.gov.uk The UK Government makes over 5,400 datasets publicly available, from all central government departments and a number of other public sector bodies and local authorities. The site also includes links to SPARQL tutorials and examples:
`http://data.gov.uk/sparql`

DBPedia contains structured information from Wikipedia (> 100 million triples):
`http://dbpedia.org/sparql`, see `http://www.dbpedia.com` for further information and documentation

DBTune provides access to music-related structured data with more than 14 billion RDF triples. The interface also allows for selecting an entailment regime that is to be used (RDF, RDFS, plus the non-standardized RDFSLite and p2r) :
`http://dbtune.org/jamendo/store/user/query`

CKAN is a platform to share, use, and find data that is publicly available
`http://semantic.ckan.net/sparql/`

Linked Movie Database A semantic web database for movies, including a large number of interlinks to several datasets on the open data cloud and references to related webpages
`http://data.linkedmdb.org/` and `http://data.linkedmdb.org/sparql`

SPARQL Editor Talis hosts a SPARQl Editor with Examples for Space Data `http://api.talis.com/stores/space/items/tutorial/spared.html`

Semantic Web Dog Food contains data about authors and publications for several conferences:
`http://data.semanticweb.org/snorql`

SPARQL Endpoint Status collects uptime information for SPARQL endpoints from CKAN
`http://labs.mondeca.com/sparqlEndpointsStatus/index.html`

7.2 RDFS

Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary ter Horst, H.J.: Journal of Web Semantics 3(2–3), 79–115 (2005)

7.3 OWL & OWL Reasoning

OWL 2 Web Ontology Language: Primer Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S. (eds.). W3C Recommendation (2009), available at `http://www.w3.org/TR/owl2-primer/`

OWL 2: The next step for OWL Cuenca Grau, B. Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., and Sattler, U.: Journal of Web Semantics: Science, Services and Agents on the World Wide Web, 6(4):309–322, 2008.

From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: Journal of Web Semantics 1(1), 7–26 (2003)

The Description Logic Handbook Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. Cambridge University Press (2003)

Hypertableau Reasoning for Description Logics Motik, B., Shearer, R., Horrocks, I.: Journal of Artificial Intelligence Research 173(14), 1275–1309 (2009)

The even more irresistible $\mathcal{SROIQ}$ Horrocks,I., Kutz,O., Sattler,U.: Proceedings of the 10th International Conference on the Principles of Knowledge Representation and Reasoning (KR 2006). pp. 57–67 (2006)

$\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$ Kazakov, Y.: Proceedings of the 11th International Conference on the Principles of Knowledge Representation and Reasoning (KR 2008). AAAI Press/The MIT Press (2008)

A Tableau Decision Procedure for $\mathcal{SHOIQ}$ Horrocks, I., Sattler, U.: Journal of Automated Reasoning 39(3), 249–276 (2007)

Reasoning in Description Logics using Resolution and Deductive Databases Motik, B.: Ph.D. thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany (2006)

A practical OWL-DL Reasoner Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A., Katz, Y.: Pelle. Journal of Web Semantics 5(2) (2007)

Reducing OWL Entailment to Description Logic Satisfiability Horrocks, I., Patel-Schneider, P.: Journal of Web Semantics 1(4), 345–357 (2004)

Rules and Ontologies for the Semantic Web Eiter, T., Ianni, G., Krennwallner, T., Polleres, A. Reasoning Web, Fourth International Summer School 2008 Springer, 2008.

Scalable Authoritative OWL Reasoning for the Web Hogan, A., Harth, A., Polleres, A.: IJSWIS "Semantic Services, Interoperability and Web Applications: Emerging Concepts". Journal Summation Volume. To appear, 2011.

Dynamic Querying of Mass-Storage RDF Data with Rule-Based Entailment Regimes Ianni, G., Krennwallner, K., Martello, A., Polleres, A.: Proceedings of the 8th International Semantic Web Conference (ISWC 2009), LNCS, Springer-Verlag 2009.

From SPARQL to Rules (and back) Polleres, A.: Proceedings of the 16th International World Wide Web Conference, 2007.

Scalable Authoritative OWL Reasoning on a Billion Triples Hogan, A., Harth, A., Polleres, A.: Proceedings of Billion Triple Semantic Web Challenge Workshop at 7th International Semantic Web Conference, 2008.

7.4 SPARQL

Semantics and complexity of SPARQL Pérez, J., Arenas, M., Gutierrez, C. ACM Transactions on Database Systems 34(3), 1–45 (2009)

Search RDF data with SPARQL McCarthy, P.: `http://www.ibm.com/developerworks/xml/library/j-sparql/`

SPARQL Tutorial – Jena/ARQ `http://jena.sourceforge.net/ARQ/Tutorial/`

SPARQL by Example – Cambridge Semantics `http://www.cambridgesemantics.com/2008/09/sparql-by-example/`

Data Extraction & Exploration with SPARQL & the Talis platform `http://www.slideshare.net/ldodds/sparql-tutorial`

Introducing SPARQL: Querying the Semantic Web Dodds, L.: `http://www.xml.com/pub/a/2005/11/16/introducing-sparql-querying-semantic-web-tutorial.html`

7.5 SPARQL over OWL Ontologies

SPARQL Beyond Subgraph Matching Glimm, B., Krötzsch, M.: In: Proceedings of the 9th International Semantic Web Conference (ISWC 2010). vol. 6496, pp. 241–256. Springer-Verlag (2010)

SPARQL-DL: SPARQL Query for OWL-DL Sirin,E., Parsia,B.: Proceedings of the 3rd OWL Experiences and Directions Workshop (OWLED 2007) (2007)

Optimizations for Answering Conjunctive ABox Queries Sirin, E., Parsia, B.: Proceedings of the 2006 Description Logic Workshop (DL 2006) (2006)

References

1. Baader, F.: Terminological cycles in a description logic with existential restrictions. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003). pp. 325–330 (2003)
2. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005). vol. 19, pp. 364–369 (2005)
3. Baader, F., Lutz, C., Suntisrivaraporn, B.: Efficient reasoning in $\mathcal{EL}^+$. In: Proceedings of the 2006 Description Logic Workshop (DL 2006). CEUR Workshop Proceedings (2006)

4. Beckett, D., Berners-Lee, T.: Turtle – Terse RDF Triple Language. W3C Team Submission (14 January 2008), available at `http://www.w3.org/TeamSubmission/turtle/`
5. Beckett, D., Broekstra, J. (eds.): SPARQL Query Results XML Format. W3C Recommendation (15 January 2008), available at `http://www.w3.org/TR/rdf-sparql-XMLres/`
6. Boley, H., Hallmark, G., Kifer, M., Paschke, A., Polleres, A., Reynolds, D. (eds.): RIF Core Dialect. W3C Recommendation (2010), available at `http://www.w3.org/TR/rif-core/`
7. Brandt, S.: Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and—what else? In: de Mantáras, R.L., Saitta, L. (eds.) Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004). pp. 298–302. IOS Press (2004)
8. de Bruijn, J. (ed.): RIF RDF and OWL Compatibility. W3C Recommendation (2010), available at `http://www.w3.org/TR/rif-rdf-owl/`
9. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: DL-Lite: Tractable description logics for ontologies. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005). pp. 602–607. AAAI Press/The MIT Press (2005)
10. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. Journal of Automated Reasoning 39(3), 385–429 (2007)
11. Charboneau, D., Feigenbaum, L. (eds.): SPARQL 1.1 Protocol for RDF. W3C Working Draft (26 January 2010), available at `http://www.w3.org/TR/sparql11-protocol/`
12. Glimm, B., Ogbuji, C. (eds.): SPARQL 1.1 Entailment Regimes. W3C Working Draft (14 October 2010), available at `http://www.w3.org/TR/sparql11-entailment/`
13. Harris, S., Seaborne, A. (eds.): SPARQL 1.1 Query Language. W3C Working Draft (14 October 2010), available at `http://www.w3.org/TR/sparql11-query/`
14. Hayes, P.: RDF semantics. URL (2004), `http://www.w3.org/TR/rdf-mt/`
15. ter Horst, H.J.: Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. Journal of Web Semantics 3(2–3), 79–115 (2005)
16. Kazakov, Y.: Consequence-driven reasoning for horn SHIQ ontologies. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009). pp. 2040–2045 (2009)
17. Kifer, M., Boley, H. (eds.): RIF Overview. W3C Working Group Note (2010), available at `http://www.w3.org/TR/rif-overview/`
18. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to query answering in DL-Lite. In: Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR 2010). AAAI Press/The MIT Press (2010)
19. Krötzsch, M.: Efficient inferencing for OWL EL. In: Proceedings of Logics in Artificial Intelligence, European Workshop (JELIA 2010). Lecture Notes in Artificial Intelligence, vol. 6341, pp. 234–246. Springer-Verlag (2010)
20. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web Ontology Language: Profiles. W3C Recommendation (2009), available at `http://www.w3.org/TR/owl2-profiles/`
21. Motik, B., Patel-Schneider, P.F., Parsia, B. (eds.): OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C Recommendation (2009), available at `http://www.w3.org/TR/owl2-syntax/`
22. Ogbuji, C. (ed.): SPARQL 1.1 Uniform HTTP Protocol for Managing RDF Graphs. W3C Working Draft (14 October 2010), available at `http://www.w3.org/TR/sparql11-http-rdf-update/`

23. Patel-Schneider, P.F., Motik, B. (eds.): OWL 2 Web Ontology Language: Mapping to RDF Graphs. W3C Recommendation (2009), available at `http://www.w3.org/TR/owl2-mapping-to-rdf/`
24. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems 34(3), 1–45 (2009)
25. Pérez-Urbina, H., Horrocks, I., Motik, B.: Efficient query answering for OWL 2. In: Proceedings of the 8th International Semantic Web Conference (ISWC 2009). Lecture Notes in Computer Science, vol. 5823, pp. 489–504. Springer-Verlag (2009)
26. Prud'hommeaux, E., Seaborne, A. (eds.): SPARQL Query Language for RDF. W3C Recommendation (15 January 2008), available at `http://www.w3.org/TR/rdf-sparql-query/`
27. Rosati, R., Almatelli, A.: Improving query answering over DL-Lite ontologies. In: Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR 2010). AAAI Press/The MIT Press (2010)
28. Schenk, S., Gearon, P., Passant, A. (eds.): SPARQL 1.1 Update. W3C Working Draft (14 October 2010), available at `http://www.w3.org/TR/sparql11-update/`
29. Schneider, M. (ed.): OWL 2 Web Ontology Language: RDF-Based Semantics. W3C Recommendation (2009), available at `http://www.w3.org/TR/owl2-rdf-based-semantics/`
30. Williams, G.T. (ed.): SPARQL 1.1 Service Description. W3C Working Draft (14 October 2010), available at `http://www.w3.org/TR/sparql11-service-description/`